

X/O/P/E/N

PORTABILITY GUIDE

SYSTEM V SPECIFICATION  
SYSTEM CALLS & LIBRARIES

PORTABILITY GUIDE  
SYSTEM V SPECIFICATION SYSTEM CALLS & LIBRARIES

X/O/P/E/N

2



# X/O/P/E/N/I

PORTABILITY GUIDE

SYSTEM V SPECIFICATION  
SYSTEM CALLS & LIBRARIES



© 1987, The X/OPEN Group Members

*All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.*

## X/OPEN PORTABILITY GUIDE

Set of 5 Volumes

ISBN: 0-444-70179-6

Volume 1 XVS Commands and Utilities

ISBN: 0-444-70174-5

Volume 2 XVS System Calls and Libraries

ISBN: 0-444-70175-3

Volume 3 XVS Supplementary Definitions

ISBN: 0-444-70176-1

Volume 4 Programming Languages

ISBN: 0-444-70177-X

Volume 5 Data Management

ISBN: 0-444-70178-8

### *Published by:*

ELSEVIER SCIENCE PUBLISHERS B.V.

P.O Box 1991

1000 BZ Amsterdam

The Netherlands

### *Sole distributors for the U.S.A. and Canada:*

ELSEVIER SCIENCE PUBLISHING COMPANY, INC.

52 Vanderbilt Avenue

New York, N.Y. 10017

U.S.A.

*Any comments relating to the material contained in the X/OPEN Portability Guide may be submitted to the X/OPEN Group by letter via the Publisher or directly by Electronic Mail to:*

*xopen@inset.co.uk*

PRINTED IN THE NETHERLANDS





# **Contents**

## **PREFACE**

## **THE COMMON APPLICATIONS ENVIRONMENT**

## **XVS SYSTEM CALLS AND LIBRARIES**

1. Introduction
2. System Calls
3. Subroutines And Libraries
4. File Formats
5. Header Files
6. Reserved For Future Use
7. Special Files





## **Trademarks**

UNIX<sup>™</sup> is a registered trademark of AT&T in the USA and other countries.

C-ISAM<sup>™</sup> is a trademark of Informix Corporation.

LEVEL II COBOL<sup>™</sup> is a trademark of Micro Focus Limited.

XENIX<sup>™</sup> is a trademark of Microsoft Inc.

IBM<sup>™</sup> is a trademark of International Business Machines Corp.

X/OPEN<sup>™</sup> is a licensed trademark of the X/OPEN Group Members.

POSIX<sup>™</sup> is a trademark of the Institute of Electrical and Electronic Engineers Inc.





## Preface

X/OPEN represents a major breakthrough in the world of standards for the information technology industry. Ten\* of the world's major information system suppliers have come together to encourage applications portability resulting in tangible benefits for computer users, independent software houses and for the suppliers themselves.

The Group's principal aim is to increase the volume of applications available and to maximise the return on investments in software development made by Users and Independent Software Vendors.

This is achieved by ensuring portability of application programs at the source code level. Through this portability, users can mix and match computer systems and applications software from many suppliers, and thus investment in applications software is protected into the future.

In order to provide such portability, the Group defines a **Common Applications Environment** built on the interfaces to the **UNIX** operating system, as defined in the AT&T System V Interface Definition, and covering other aspects required of a comprehensive applications interface.

The X/OPEN Portability Guide contains an evolving portfolio of practical standards for application portability. All of the members of X/OPEN guarantee to support the standards defined, leading to:

- Growing portability
- No dependence on a single source - freedom of choice
- Increased application software selection
- More security in software investments
- International support for the *Common Applications Environment*

X/OPEN is not a standards-setting organisation; it is a joint initiative by members of the business community to integrate evolving standards into a common, beneficial and continuing strategy.

---

\* At the time of publication, the membership of the X/OPEN Group was BULL, DEC, ERICSSON, HEWLETT-PACKARD, ICL, NIXDORF, OLIVETTI, PHILIPS, SIEMENS and UNISYS



Issue 2 of the X/OPEN Portability Guide (published in January 1987) comprises five volumes defining the interfaces currently identified as components of the Common Applications Environment.

Volume 1	System V Specification: Commands and Utilities
Volume 2	System V Specification: System Calls and Libraries
Volume 3	System V Specification: Supplementary Definitions XVS Internationalisation XVS Terminal Interfaces XVS Inter-Process Communication XVS Source Code Transfer
Volume 4	Programming Languages C Language COBOL Language
Volume 5	Data Management Indexed Sequential Access Method (ISAM) Relational Database Language (SQL)

In addition, each volume includes an introduction giving the philosophy of the Common Applications Environment and an overview of its components.

This guide is aimed at both the decision makers and the implementation teams of:

- Independent Software Vendors
- Software Houses
- Users
- Equipment Manufacturers

The Guide is designed to sit permanently on the desk, serving as a common reference point for anyone directly concerned with the practical side of software development, namely systems designers, programmers and consultants.

The various parts of the Portability Guide are closely interrelated. Any reference from one part of the definition to another part uses the title of the other part as a reference (e.g., "XVS TERMINAL INTERFACES").



# **Acknowledgements**

X/OPEN gratefully acknowledges:

- **AT&T** for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.
- The **/usr/group** Standards Committee, whose Standard contributed to the Group's work.
- **Informix Corporation.** of Menlo Park, California (Telex no. 361834) for permission to use material from the specification of their C-ISAM product and for provision of that material in machine readable form.
- **Micro Focus Ltd.** of Newbury, Berkshire for permission to use material from the specification of their LEVEL II COBOL compiler.
- The assistance given by the following companies in the preparation of the Database Language (SQL) definition:

Informix Corporation  
Oracle Corporation  
Queensland Information Technology  
Relational Technology Inc.  
Unify Corporation



## ***Referenced Documents***

The following documents are referenced in this guide:

- System V Interface Definition (Spring 1985 - Issue 1)
- System V Interface Definition (Spring 1986 - Issue 2)
- UNIX System V Release 2.0 Programmer's Reference Manual (April 1984 - Issue 2)
- UNIX System V - Release 2.0 Programming Guide (April 1984 - Issue 2)
- ANS Draft Proposal for C Language (October 1986 - ANS X3J11/86-151)
- 1984 /usr/group Standard
- IEEE P1003.1 Trial Use Standard (April 1986)
- Informix Corporation C-ISAM Reference Manual (Version 2.10 - January 1985)
- MicroFocus Level II COBOL Language Reference Manual (Version 2.5 and 2.6, Issue 7 - April 1984)
- Standard for COBOL (ANS X3.23-1974)
- Standard for COBOL (ANS X3.23-1985)
- Standard for FORTRAN (ANS X3.9-1978)
- Standard for Database Language (SQL) (ANS X3.135-1986)
- Standard for PASCAL (ISO 7185-1983)



# X/O/P/E/N/

PORTABILITY GUIDE

THE COMMON APPLICATIONS  
ENVIRONMENT







# **Contents**

Chapter	1	THE COMMON APPLICATIONS ENVIRONMENT
Chapter	2	SYSTEM V
	2.1	INTRODUCTION
	2.2	THE EVOLVING STANDARD
	2.2.1	Origins
	2.2.2	The IEEE "POSIX" Standard
	2.2.3	The AT&T System V Interface Definition
	2.3	THE X/OPEN SYSTEM V SPECIFICATION
	2.3.1	System Calls and Libraries
	2.3.2	Inter-process Communication
	2.3.3	Commands and Utilities
Chapter	3	INTERNATIONALISATION
	3.1	INTRODUCTION
	3.2	The X/OPEN NATIVE LANGUAGE SYSTEM
Chapter	4	C LANGUAGE
	4.1	INTRODUCTION
	4.2	C LANGUAGE PORTABILITY GUIDELINES
	4.3	THE ANS X3J11 DRAFT STANDARD
	4.4	THE C PROGRAM PORTABILITY CHECKER ( <i>lint</i> )
Chapter	5	OTHER PROGRAMMING LANGUAGES
	5.1	INTRODUCTION
	5.2	COBOL
	5.3	FORTRAN
	5.4	PASCAL
Chapter	6	DATA MANAGEMENT
	6.1	INTRODUCTION
	6.2	INDEXED SEQUENTIAL ACCESS METHOD (ISAM)
	6.3	RELATIONAL DATABASE LANGUAGE (SQL)



<b>Chapter</b>	<b>7</b>	<b>SOURCE CODE TRANSFER BETWEEN MACHINES</b>
	7.1	INTRODUCTION
	7.2	FLOPPY DISC STANDARD
	7.3	MAGNETIC TAPE
	7.4	UTILITIES
<b>Chapter</b>	<b>8</b>	<b>NETWORKING AND COMMUNICATIONS</b>
	8.1	NETWORKING AND COMMUNICATION
	8.2	OPEN SYSTEMS INTERCONNECTION
	8.3	GENERALISED INTER-PROCESS COMMUNICATION, IPC
	8.4	DISTRIBUTED FILE SYSTEM
	8.5	DISTRIBUTED TRANSACTION PROCESSING



# **The Common Applications Environment**

The formation of the X/OPEN Group represents a major initiative by an international group of suppliers of computer systems to create a free and open market, offering Independent Software Vendors (ISVs) as wide a market as possible for their products and giving users an increased return on investment in application software.

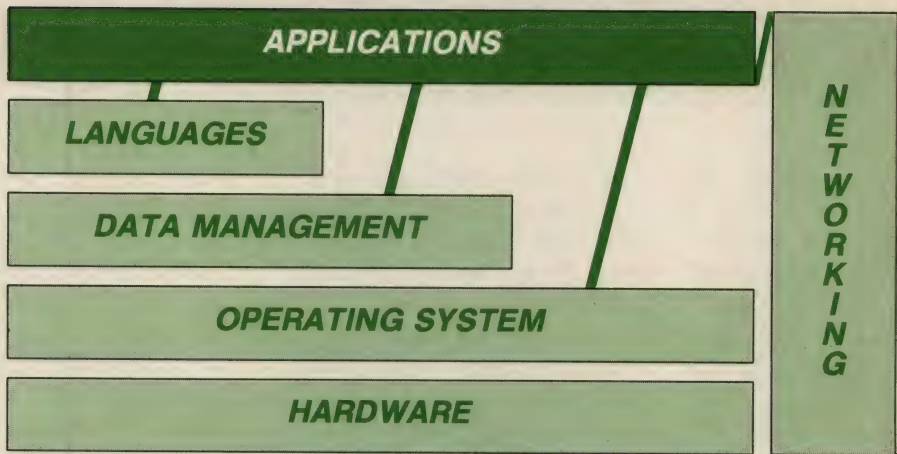
The current dominance of proprietary machine environments is restricting the growth of the computer industry. Users tend to get locked into a particular proprietary system by the investment they have made in the applications. Independent Software Vendors are discouraged from writing applications for a particular environment because of the limited markets caused by this fragmentation. This means that there is very little generally available software for each type of system, thus increasing the size of investment needed by each user. All this in turn limits the sales potential of machines from the computer suppliers.

The objective shared by the members of the X/OPEN Group is to establish a Common Applications Environment to the mutual advantage of users, Independent Software Vendors and computer suppliers. Applications written to operate in this environment will be portable at the source code level to a wide range of machines, thereby releasing the user from dependence on a single supplier, reducing the necessary investment in applications, considerably increasing the market for independent software and opening up the market for systems suppliers.

The existence of these "Open Systems" allows users to mix and match systems from different suppliers, and to move applications between machines to meet changing requirements as business grows, thereby giving protection of investment in applications software into the future.

The great increase in the potential market encourages the Independent Software Vendors to produce a wealth of general applications packages, and the availability of this further reduces the investment needed by the users. The whole situation is thus mutually reinforcing.





The foundations of the Common Applications Environment are the interfaces of the UNIX System V operating system, as defined in the AT&T "System V Interface Definition", and the C language.

To define a complete environment for portable applications, it is also necessary to satisfy the requirements for data management, integration of applications, data communications, distributed systems, the use of high level languages and the many other aspects involved in providing a comprehensive applications interface. The X/OPEN Group intends, therefore, to publish progressively definitions covering these areas.

The systems of the X/OPEN Group members that support interfaces derived from UNIX operating systems will do this according to the X/OPEN definitions and will support the full Common Applications Environment.

A specific Common Applications Environment feature may not, however, be present if it is not relevant in the market area in which a particular system is sold. For example, a system sold only in a scientific context might not support COBOL. Conversely, a particular system may support features over and above those of the Common Applications Environment, some of which may partially overlap. An example of this could be that an alternative dialect of COBOL is supported in addition to that of the Common Applications Environment.

The X/OPEN Group is primarily concerned with standards selection and adoption. The general policy is to use International Standards, where they exist, and to adopt "de facto" standards in other cases.

Where International Standards do not exist, it is X/OPEN policy to work closely with standardisation bodies to encourage their emergence.



## *The Common Applications Environment*

It is important that the defined elements of the Common Applications Environment be readily achievable on member systems, and have wide acceptance. For this reason, the definitions, in general, fall within the capabilities of at least one currently available popular product.

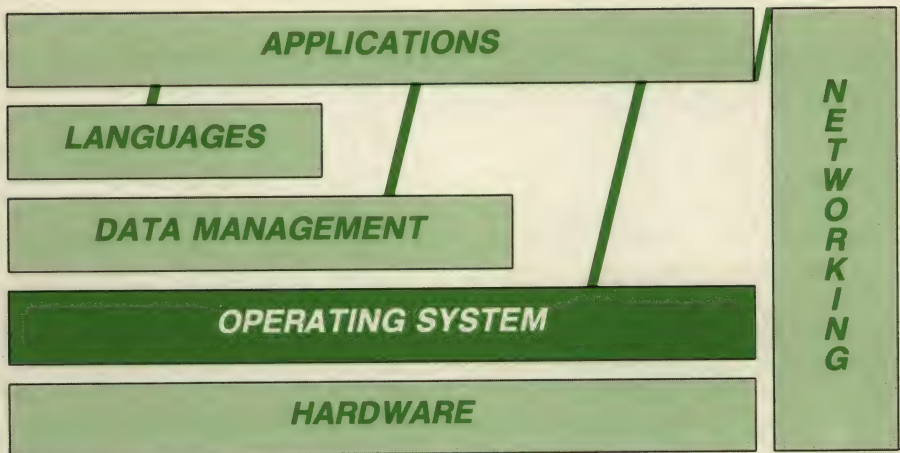
In this guide, certain aspects of the Common Applications Environment are defined with reference to the interfaces offered by specific products. This does not mean that member systems will necessarily contain these products, but that the defined interfaces will be supported. Indeed the method of support for an interface on a particular system may change with time.







# **X/OPEN System V Specification (XVS)**



## 2.1 INTRODUCTION

The X/OPEN System V specification (XVS) defines the applications interfaces provided by the underlying operating system and forms the foundation of the Common Applications Environment.

The XVS is derived from a series of standards activities. The evolving standard is briefly addressed, and the relationship between the X/OPEN System V Specification and the System V Interface Definition and other standards is explained.



## 2.2 THE EVOLVING STANDARD

### 2.2.1 Origins

The UNIX operating system was developed by Ritchie and Thompson at Bell Laboratories in the early 1970s. The current AT&T System V version may be traced back directly to that first system.

For many years, it remained basically an academic product. More recently, computer suppliers have adopted the UNIX system as a multi-tasking, multi-user and portable operating environment. They have based their systems on one of several releases, variants or look-alikes. Of these, the most widely used were Version 7, System III, the Berkeley system and XENIX.

Although these systems had much in common, the degree of compatibility at the application interface level was insufficient to permit the development of totally portable applications.

### 2.2.2 The IEEE "POSIX" Standard

/usr/group, a group of users of UNIX derivatives in the USA, established a committee with the objective of proposing a set of standards for application level interfaces. After publishing its standard, together with a reviewer's guide, the group decided to seek IEEE status for the standard. In late 1984, the /usr/group standards committee closed its activities in its own name and its members were encouraged to become involved in the IEEE group, known as P1003.

The P1003 group published a "trial-use" standard in early 1986, which has the status of a "Draft American National Standard". This "Portable Operating System for Computer Environments" (POSIX) is expected to be revised and submitted for approval in 1987.

The IEEE P1003 group is working to extend the POSIX standard. It is expected that the next area to be standardised will be the subset of commands which offer an interface to applications.



### 2.2.3 The AT&T System V Interface Definition

The "System V Interface Definition" (SVID), first published by AT&T in the spring of 1985, represented a major standards initiative. AT&T were prominent in the activities of /usr/group and the influence of /usr/group can clearly be seen in the SVID. The stated purpose of the SVID is to define common interfaces for all System V implementations.

The definition groups interfaces into a mandatory *base* plus a series of *extensions*. The *base* interfaces must be present in any implementations of System V. If any interface from an *extension* is supported, it must adhere to the definition.

Issue 1 of the SVID comprised a single volume defining operating system interfaces (known as *system calls* and *library routines*) available to applications as directly called external functions and defined in terms of invocation from C-language programs.

Issue 2 of the SVID was published in early 1986 and comprised two volumes. The first volume contained the same material as issue 1, with some restructuring to improve ease of use and some changes to correct errors. It comprised the *Base System Definition* plus a single extension referred to as the *Kernel Extension*.

The second volume primarily defined commands and utilities, normally invoked through a command interpreter. It comprised further extensions referred to as the *Base Utilities Extension*, *Advanced Utilities Extension*, *Administered Systems Extension*, *Software Development Extension*, and *Terminal Interface Extension*. The latter two include library routines in addition to utilities.



## 2.3 THE X/OPEN SYSTEM V SPECIFICATION

The X/OPEN System V Specification (XVS) is based upon the AT&T System V Interface Definition, but also taking into account the trial use standard published by IEEE and the capabilities of the existing AT&T System V product.

The XVS is organised into a number of self-contained sections:

"XVS SYSTEM CALLS AND LIBRARIES" defines the Operating System Interfaces and broadly corresponds to Volume 1 of the SVID.

"XVS COMMANDS AND UTILITIES" defines commands and utilities and broadly corresponds to Volume 2 of the SVID. The purpose of the X/OPEN Portability Guide is to facilitate the portability of applications. As such, system administration is outside of its scope and the routines included in the AT&T *Administered System Extension* are not defined.

"XVS TERMINAL INTERFACES" defines a set of portable interfaces to locally connected asynchronous terminals and broadly corresponds to the AT&T Terminal Interface Extension in Volume 2 of the SVID.

"XVS INTER-PROCESS COMMUNICATION" defines interfaces to shared memory, semaphores and message passing, included as an interim mechanism to satisfy the immediate requirements of Inter-Process Communication facilities.

### 2.3.1 System Calls and Libraries

"XVS SYSTEMS CALLS AND LIBRARIES" contains a full definition of interfaces to system calls and library routines and broadly corresponds to Volume 1 of the SVID.

The X/OPEN Group has extended the SVID in a number of areas:

- Certain changes have been included, which the SVID denotes as future directions.
- The use of symbolic names to replace numeric constants, introduced by AT&T in their SVID, has been extended.
- Clarification of existing wording has been introduced in a limited number of places to "tighten" the specification.
- The opportunity has been taken to correct clerical errors in the SVID.
- Definitions have been included of a number of further UNIX System V Release 2.0 functions which are in widespread use by application developers.

The relationship between the XVS and the SVID is clearly stated. The whole of the SVID *base* definition is included as mandatory with the exception of the maths group, which is not mandatory for systems sold into markets where it is not relevant. *Termio* was not mandatory in issue 1 of the XVS because of some difficulties in implementation. These have now been resolved, and the routine is now mandatory, except in systems which do not support locally connected asynchronous lines.



The XVS incorporates all the interfaces within the SVID *kernel extension set* although a number are defined as optional.

In the XVS, interfaces defined as *optional* will be available on most but not necessarily all X/OPEN systems; use of them could restrict portability. Any *optional* interface supported on an X/OPEN system will conform to the X/OPEN specification.

The XVS defines interfaces in terms of their interface syntax and run-time behaviour, without constraining the method of their implementation. The names "system calls" and "subroutines" are retained purely for compatibility with other documentation.

### 2.3.2 Inter-process Communication

The kernel extension interfaces relating to shared memory, semaphores and message passing are included in "XVS INTER-PROCESS COMMUNICATION" as a short-term mechanism to satisfy the immediate requirements for Inter-Process Communication facilities. However, they are machine specific and cannot be supported on all hardware architectures. The Group believes that a more generalised approach to the whole subject of Inter-Process Communication is required.

### 2.3.3 Commands and Utilities

"XVS COMMANDS AND UTILITIES" contains a full definition of interfaces to commands and utilities and broadly corresponds to Volume 2 of the SVID. The interfaces are split functionally into those which are intended to provide an applications interface (referred to as *Standard Utilities*) and those which are only intended to be used by development programmers or during the porting of applications to an X/OPEN system (referred to as *Development Utilities*). The Standard Utilities will be present in all X/OPEN systems, as they are needed to provide a run-time interface to applications. The Development Utilities may only be present in development systems; their respective descriptions are clearly annotated to indicate this. This same distinction is also present in the SVID. The X/OPEN development utilities correspond exactly to the SVID *Software Development Extension*.

The definition of standards for commands and utilities is an evolving process and X/OPEN intends to participate fully.

The current definition is a valuable first step, but additional work will be needed to evolve towards a complete standard. For example:

- The number of options defined for many of the commands is excessive and includes functionality which is rarely used or is implementation specific.
- There are too many different ways of achieving the same results.
- Many of the current descriptions were written to record the observed behaviour of already existing utilities and the level of precision is inadequate for use as a definitive standard.



Rectification of this is an enormous task and the current X/OPEN definition is of necessity based on the available documentation, but incorporates extensive annotation to highlight potential portability problems resulting from the points listed above. To achieve maximum portability, application developers should avoid the use of the functionality so annotated.

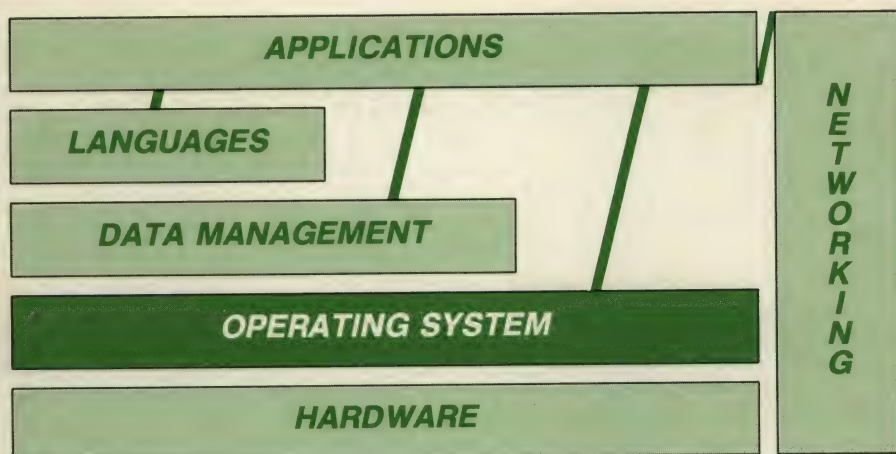
X/OPEN is working on a substantially improved definition of commands, with the number of options reduced to those in common use and with a higher level of specification. In addition to the current X/OPEN specification, "XVS COMMANDS AND UTILITIES" contains a proposed template for improved specifications together with a number of examples.

This improvement process will be carried out in close consultation with the various user organisations and standards bodies, such as IEEE and ISO, to ensure that the result is a single standard definition of operating system commands and utilities.

Readers of the X/OPEN Portability Guide are invited to participate directly in the consultative process to ensure that the evolving standard matches the requirements of existing and future applications.



## Internationalisation



### 3.1 INTRODUCTION

X/OPEN members market systems in many countries. Our customers and users speak many different languages and conform to different cultural conventions and business practices. It is important therefore that X/OPEN systems are capable of supporting a range of language and cultural environments. In many cases a strong requirement also exists to cope with these variations on the same system. An example is within the administration of the European Economic Community.

To date UNIX operating systems and most systems derived from them have been based on the ASCII 7-bit coded character set and on American English. There are no facilities for dealing with other coded character sets, nor for supporting different languages and cultural conventions.

The requirement for effective mixed language working brings with it the need for coded character sets larger than can be accommodated by 7-bit characters, as does the requirement to support the more complex languages. At the same time there is a trade-off between the ability to handle larger coded character sets, and the amount of storage required to hold the data. For most European requirements an 8-bit system provides the correct balance. For the major Eastern languages (such as Chinese and Japanese) a 16-bit system is necessary, even to support a single language.



To satisfy these requirements enhancements must be made to the system to provide full data transparency to applications, allowing flexibility in the choice of coded character set(s) employed. Additionally, the system must allow program messages (both input and output) to be handled in the native language of each user, as well as providing cultural dependent data items (such as date formats and currency symbols).

### 3.2 THE X/OPEN NATIVE LANGUAGE SYSTEM

The X/OPEN Native Language System (NLS) is a set of interfaces designed to facilitate the development of applications that can operate in many different language and cultural environments. The interfaces have been derived from those of the Native Language Support system developed by the Hewlett-Packard Company of Palo Alto, California. They have been further enhanced by X/OPEN and have been modified in strategic areas to more closely relate to the Internationalisation proposals of the Draft Proposed American National Standard for the C Programming Language.

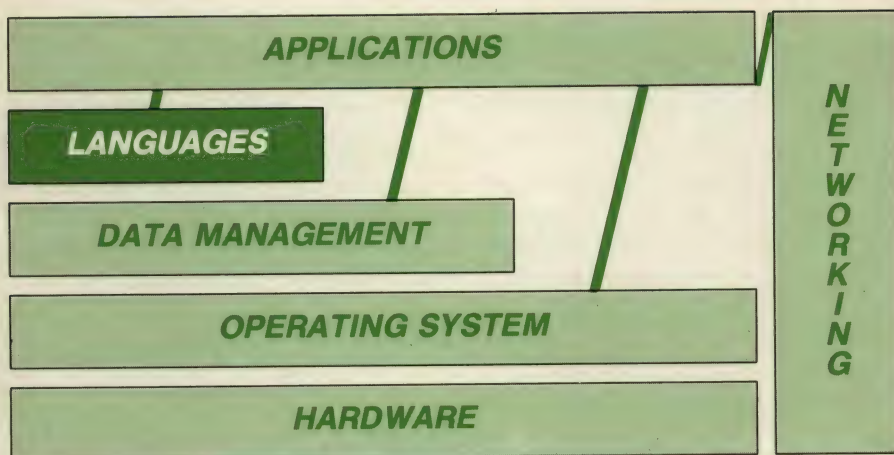
The first issue of the specification, defined in "XVS INTERNATIONALISATION", concentrates on facilities for the development of internationalised applications (rather than on internationalising the operating system itself), and on the 8-bit coded character set situations.

The following groups of facilities are defined:

- A message catalogue system which allows program messages to be held apart from the program logic, translated into different native languages, and the appropriate version retrieved by the program at run time.
- An announcement mechanism whereby native language, local custom (territory) and codeset requirements appropriate to each user can be identified to applications at run time.
- Enhanced interface definitions of standard C library functions, which provide language dependent character type classification, upper to lower case and lower to upper case character conversions, date and time messages, floating point to string conversions, and text collation.
- Library functions which allow programs to determine cultural and language specific data dynamically (e.g. the format of date and time strings, weekday and month names, currency symbols, etc.).
- A set of standard commands and library functions which will operate correctly with 8-bit characters.



# C Language



## 4.1 INTRODUCTION

This chapter addresses the C language and guidelines for portability when writing C code.

Currently the American National C Language Standards committee, X3J11, is working towards a standard for the C programming language. The X/OPEN Group is represented on that committee by member companies and intends to adopt the standard, once it has been established as a practical reality.

Meanwhile, the X/OPEN definition included in "C LANGUAGE" is based upon that given in Chapter 2 of the "System V Programming Guide", Release 2.0, published by AT&T.

## 4.2 C LANGUAGE PORTABILITY GUIDELINES

Whilst the C language provides the basis for applications portability, it is easy to write statements, using valid C constructs, that are machine specific. Care has to be taken when writing programs that are intended to be portable across a range of systems. "C LANGUAGE" includes advice towards ensuring portability.

#### 4.3 THE ANS X3J11 DRAFT STANDARD

The ANS X3J11 standard has been published in a draft form but may still change before it is approved. However, it is already clear that the standard will impose certain restrictions such that programs written to the current C language definition may not work correctly, if the source is later passed through a compiler that supports the ANS standard.

To address this, "C LANGUAGE" includes advice on writing programs to avoid these problems.

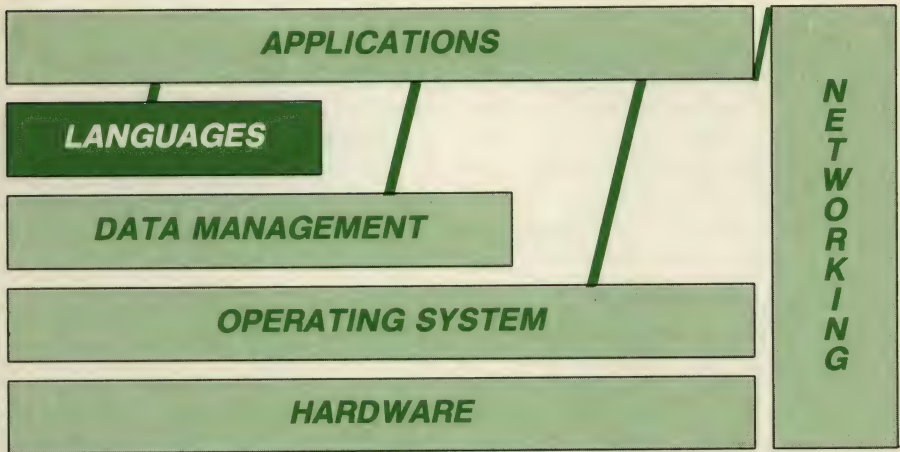
#### 4.4 THE C PROGRAM PORTABILITY CHECKER (*lint*)

The *lint* program checks C source programs for violation of most of the portability rules. It also gives a more stringent enforcement of the type rules of C than is provided by most C compilers. A further option detects a number of wasteful or error-prone constructions which nevertheless are syntactically correct.

Use of the *lint* program is recommended: it is described in "C LANGUAGE".



## Other Programming Languages



### 5.1 INTRODUCTION

This chapter addresses programming languages other than C on X/OPEN systems. It covers the inclusion of the principal high level languages in the Common Applications Environment.

To date, The X/OPEN Group has established definitions for COBOL and FORTRAN and PASCAL.

## 5.2 COBOL

The X/OPEN COBOL definition identifies a common set of language facilities that will be supported by COBOL compilers on all member systems. Applications written to this definition will be portable to any X/OPEN system.

The ISO Working Group and ANS COBOL committee have been working for some years towards a revised standard for COBOL to reflect more accurately the capabilities of modern COBOL compiling systems. The latest international standard, "X3.23 - 1985" was approved during 1985. At the time of publication of Issue 2 of the Portability Guide, there are few compilers in compliance with the revised standard and hence the X/OPEN group feels that major changes in the X/OPEN definition to reflect the new standard would be premature. It is likely, however, that any future edition of the X/OPEN COBOL language definition will relate to "COBOL 1985" and the new standard has been used as a reference when eliminating obsolete elements from the COBOL definition.

The most widely followed standard for COBOL is still that defined in the earlier 1974 Standard, "ANS X3.23-1974", to which most current COBOL compilers substantially conform.

The 1974 standard is incomplete in the area of facilities for interaction with the on-line user. To overcome this deficiency, most COBOL compilers provide extensions to the *ACCEPT* and *DISPLAY* verbs, but they do this in incompatible ways. Since the majority of applications now include interactive operation, it is necessary for a standard form of *ACCEPT* and *DISPLAY* to be defined in the X/OPEN Common Applications Environment.

In order to have an X/OPEN definition that is achievable on member systems within a short timescale, and one that would have immediate widespread acceptance, it has been based on the definition of COBOL embodied in a popular product: Micro Focus LEVEL II COBOL, which itself conforms to the "ANS X3.23-1974".

The Micro Focus LEVEL II COBOL language specification includes a number of other extensions beyond the 1974 standard, in addition to those to *ACCEPT* and *DISPLAY*. None of these are currently included in the X/OPEN definition. The X/OPEN definition also applies restrictions to the ANS-based parts of the LEVEL II definition.

Whilst the X/OPEN COBOL definition is based on the specification of a particular product, the means of implementation across the systems of the X/OPEN members may vary. Any particular system may support extensions beyond the facilities identified, but their use is likely to impede portability.

The X/OPEN COBOL definition is given in detail in "COBOL LANGUAGE" and its relationship to the 1974 standard is clearly shown.

The definition is given in terms of the command syntax derived from the LEVEL II COBOL Reference Manual. The semantics of the *ACCEPT* and *DISPLAY* verbs are defined in "COBOL LANGUAGE". The semantics of all other elements of the language are defined by the "X3.23-1974" standard.



### 5.3 FORTRAN

The X/OPEN definition for FORTRAN is the formal definition given in the American National Standards document "FORTRAN 77, ANS X3.9 - 1978". This has had wide-scale acceptance throughout the world and there are many certified compilers available.

The majority of FORTRAN compilers, while adhering to the basic FORTRAN 77 standard, also offer extensions beyond that standard. There is little compatibility in these extensions between compilers and they do not form part of the X/OPEN definition. Developers are warned that use of these extensions will affect the portability of FORTRAN programs.

### 5.4 PASCAL

The current X/OPEN definition for PASCAL is the formal definition given in the International Organisation for Standardisation document "Programming Languages - PASCAL" ISO 7185-1983 (level 1).

This is well accepted throughout the world and there are significant numbers of certified compilers available.

In order to enhance the portability of programs among PASCAL implementations on X/OPEN systems, the X/OPEN Group has decided to give a uniform definition for certain features designated as implementation-defined in ISO 7185.

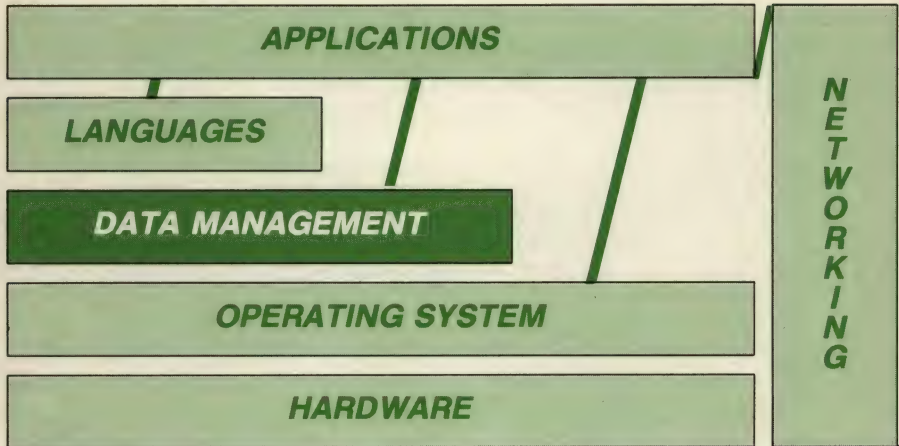
- When the required identifiers "input" and "output" occur as program parameters they shall be bound by default to the external system files STDIN and STDOUT respectively.
- There shall be no implementation dependent restrictions on the base-type of a set-type which disallow 0 as the minimal ordinal value and 255 as the maximum ordinal value of that base-type.
- (lazy input) the underlying data transfer action required for a call to the predefined procedure "get" for a textfile (both explicit and where implied by "reset" and "read"), shall be postponed until one of the following events, if any, whichever occurs first :
  - a. access to the buffer-variable of the file
  - b. the buffer-variable of the file is passed as an actual variable parameter to a procedure or function
  - c. a call to the predefined function "eoln" for that file
  - d. a call to the predefined function "eof" for that file

The majority of Pascal compilers also offer extensions beyond the current ISO Standard PASCAL definition. There is little compatibility in these extensions between compilers and developers are warned that use of these extensions may affect the portability of PASCAL programs.





# Data Management



## 6.1 INTRODUCTION

The input/output facilities supported by System V consist only of byte-stream read and write operations on files. No facilities are provided for operating on files as sets of records. This leads to application writers having to make their own arrangements for record handling, resulting in both a multiplication of effort and a proliferation of non-standard methods.

Data Management is a key element in the integration of applications. Applications written in a variety of languages must be able to work on the same basic data in the same form, and data must be passed easily and efficiently between applications.

Addressing these issues, the X/OPEN Group defines interfaces for the creation, management and manipulation of indexed files, generally known as the Indexed Sequential Access Method (ISAM) and for access to relational database management systems, the standard Relational Database Language (SQL).

The availability of these interfaces on X/OPEN systems will not only provide application portability, but will ease and encourage integration.

## 6.2 INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

The X/OPEN definition for ISAM, which is contained in "INDEXED SEQUENTIAL ACCESS METHOD", is a major subset of the specification of the C-ISAM product, Version 2.10, published by Informix Corporation

The full specification of C-ISAM contains implementation details specific to that product, in addition to the definition of the interface available to applications. Only the applications interface forms part of the X/OPEN definition; implementation details specific to C-ISAM have been omitted. Indeed, there are alternative implementations available on particular member systems.

## 6.3 RELATIONAL DATABASE LANGUAGE (SQL)

To reflect the growing significance of Relational Data Base systems, the X/OPEN group has defined application interfaces embedded within high level "host" languages to a relational database management system for a free-standing database.

The widely accepted standard for access to relational data base is that defined in the American National Standard document Relational Database Language (SQL) "ANS X3.135-1986".

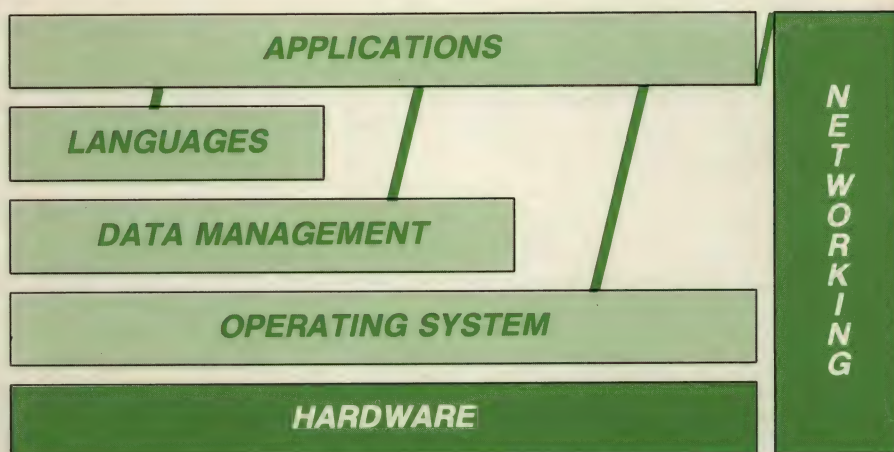
The X/OPEN definition is based closely on "X3.135-1986" but taking careful account of the capabilities of the leading relational database management systems currently available. The X/OPEN group has worked closely with the vendors of these products throughout.

"X3.135-1986" allows for two levels of compliance, Level 1 and Level 2. Most existing products comply only at Level 1 although it is expected that a significant number of products will have achieved full compliance with Level 2 before the end of 1987. "X3.135-1986" Level 1 SQL is not an adequate definition for application developers, since it leaves too many areas as implementor-defined. In preparing its definition, the X/OPEN group has examined these areas carefully and an agreed X/OPEN approach defined.

The X/OPEN SQL definition is contained in "RELATIONAL DATABASE LANGUAGE (SQL)", and contains a full description of the syntax and semantics of SQL together with a detailed comparison between the X/OPEN definition and the "ANS X3.135-1986" standard.



# Source Code Transfer Between Machines



## 7.1 INTRODUCTION

One of the major problems inhibiting the porting of applications between UNIX system derivatives is that of incompatible media standards and the physical problems of transferring source code in machine readable form.

The X/OPEN Group takes this problem seriously and has agreed common standards for the transfer of source code. Detailed standards are defined in "SOURCE CODE TRANSFER".

Standards are defined for transfer of 5¼" floppy discs and ½" magnetic tape between machines. Because of the different nature of X/OPEN systems, ranging from single user work stations to large mainframes, it is not possible to define formats which are portable across the whole range. Defining standards for both floppy discs and ½" magnetic tape gives the highest practical coverage of systems.

Current differences in the physical recording formats between cartridge tape devices prevents the definition of a standard for this popular medium.

Because of restrictions imposed by existing hardware, some X/OPEN members are not able to support the floppy disc standard.

## 7.2 FLOPPY DISC STANDARD

As exchange media, the X/OPEN group defines standards for 40 and 80 track floppy discs. It is intended that the prime format should be 80 track, with 40 track retained for compatibility with personal computers. X/OPEN systems equipped only with 80 track disc drives will offer the facility to read 40 track floppy discs by skipping alternate tracks.

## 7.3 MAGNETIC TAPE

The X/OPEN standard for magnetic tape covers ½" magnetic tape, with a number of different recording formats and densities. The prime format is 9 track Phase Encoded at 1600 bits per inch.

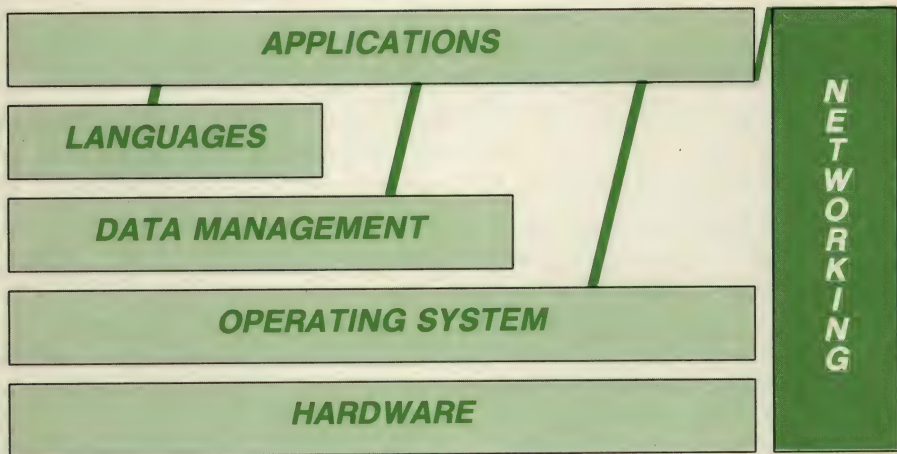
## 7.4 UTILITIES

"SOURCE CODE TRANSFER" includes the definition of two alternative utilities for the archiving of files to the transfer medium and their subsequent retrieval, *tar* and *cpio*.

In addition, guidelines are given on the use of direct machine to machine connection and the *uucp* utility, as a means of transferring files between X/OPEN systems.



# Networking and Communications



## 8.1 NETWORKING AND COMMUNICATION

The general target for computer data communications is interworking between systems of different types from different suppliers. Future X/OPEN definitions for such open systems interworking can be expected to embrace the ISO OSI standard.

Two services specific to systems supporting the System V Interface have been identified. These are "Generalised Inter Process Communication" (IPC) and "Distributed File System".

Many current commercial applications are supported via interactive transaction processing systems. X/OPEN definitions in this area can be expected to be in line with emerging International Standards.

## 8.2 OPEN SYSTEMS INTERCONNECTION

For interworking to be possible, systems must have common methods of describing both tasks and data, and must be capable of functioning in a defined manner. To describe interconnection, an architectural model is required. The International Organisation for Standardisation, ISO, has developed the "Reference Model for Open Systems Interconnection" (IS7498). This is often referred to as the "ISO OSI model" or the "ISO 7-layer model". This model sets a framework into which protocol and service standards can be set.

The ISO OSI target is to have a complete set of protocol and service definitions which comply with the 7-layer model and which are internationally agreed and published as ISO standards.

A complete set of ISO OSI standards does not yet exist. Where standards are available, they contain options. For practical systems to be built, it is necessary to have a very clear definition of standards to be adopted and the options to be used.

To provide a clear statement of the standards to be used, a specialist working group has been formed by the 12 major European vendors who propose the technical objectives for "ESPRIT", the "European Strategic Programme for Research into Information Technology". This is called the "Standards Promotion and Application Group, SPAG".

Where ISO standards do not yet exist, interim standards from one of the national or international standards bodies are adopted by SPAG. Such standards are expected to form the basis of future ISO standards. SPAG is not itself a standard making body. Its recommendations will reflect evolving standards.

X/OPEN member companies are committed to the ISO OSI target and the adoption of ISO standards. The group will monitor SPAG recommendations.

X/OPEN intends to define application interfaces for access to OSI services to ensure the portability of applications and library routines.



### 8.3 GENERALISED INTER-PROCESS COMMUNICATION, IPC

UNIX operating systems provide limited IPC capabilities in the form of "pipes" and "fifos". Kernel extensions within the AT&T System V Interface Definition provide some further IPC mechanisms for the passing of messages between processes in the same memory address space. These extensions were omitted from issue 1 of the X/OPEN definition because it was believed that a much more generalised mechanism for peer to peer communication between processes, either in the same physical machine or in different machines connected via some communications medium is needed.

The X/OPEN group is working on the definition of such a mechanism and but in the short term, it is recognised that there are some applications which need access to such IPC capabilities as currently exist. "XVS INTER-PROCESS COMMUNICATION" gives detailed definitions for

- Message passing between processes.
- Shared memory.
- Semaphores.

It must be recognised that these routines cannot be implemented on all hardware architectures and hence are optional in the X/OPEN definition. However, where the interfaces are supported, the behaviour will be as defined.

#### 8.4 DISTRIBUTED FILE SYSTEM

There is an increasing requirement to be able to access data contained within UNIX File Systems on machines connected together by a local area network from any system on that network. The totality of data accessible in this way can be regarded as a "distributed file system".

The X/OPEN Group regards the way in which local resources are made available to other systems to be a matter of system administration, and does not intend to publish detailed definitions.

The only aspect of such systems which is relevant to the application developer is the behaviour of the system towards applications at run-time.

The characteristics of any distributed file system supported by X/OPEN systems are:

- Access to the distributed file system is via the standard input/output system calls, and is identical for local and remote files.
- No changes are necessary to existing applications. Binary copies of existing applications are able to access a distributed file system, subject to the requirement that the data within a file is in a compatible format.
- Naming of remote items follows the same syntax as for local items and no new naming conventions are required.
- File locking applies across the network.



8.5 DISTRIBUTED TRANSACTION PROCESSING

Many commercial applications require interactive transaction processing facilities and the X/OPEN Group considers the provision of such facilities to be of key importance.

International Standards Organisations have not made the expected progress in this area.

The X/OPEN Group intends to take action to improve this situation.





# X/O/P/E/N/

PORTABILITY GUIDE

THE X/OPEN SYSTEM V SPECIFICATION  
SYSTEM CALLS AND LIBRARIES





# **Contents**

Chapter	1	INTRODUCTION
	1.1	OVERVIEW
	1.1.1	Rationale
	1.1.2	Contents
	1.2	STATUS OF INTERFACES
	1.2.1	Mandatory
	1.2.2	Optional
	1.2.3	Internationalisation
	1.2.4	Relationship to SVID
	1.2.5	Subject to Change
	1.3	FORMAT OF ENTRIES
	1.4	DEFINITIONS
	1.4.1	Character and Block Special Files
	1.4.2	Character Sets
	1.4.3	Command Interpreter
	1.4.4	Directory
	1.4.5	Effective User ID and Effective Group ID
	1.4.6	Fifo Special Files
	1.4.7	File Access Permissions
	1.4.8	File Descriptor
	1.4.9	File Name
	1.4.10	Parent Process ID
	1.4.11	Path Name and Path Prefix
	1.4.12	Process Group ID
	1.4.13	Process Group Leader
	1.4.14	Process ID
	1.4.15	Real User ID and Real Group ID
	1.4.16	Root Directory and Current Working Directory
	1.4.17	Special Processes
	1.4.18	Super-user
	1.4.19	Terminal Group
	1.4.20	Tty Group ID
	1.5	SIGNALS
	1.6	DIRECTORY STRUCTURE
	1.7	ENVIRONMENTAL VARIABLES
	1.8	SYSTEM RESIDENT DATA FILES
	1.9	SPECIAL FILES

- 1.10 CAVEATS
  - 1.10.1 Null Pointers
  - 1.10.2 Termio(7)
  - 1.10.3 Process IDs
  - 1.10.4 The files <values.h> and <limits.h>
- 1.11 ERRORS AND EXCEPTIONS
- 1.12 LIST OF SERVICES

## Chapter 2 SYSTEM CALLS

*access*(2)  
*acct*(2)      OPTIONAL  
*alarm*(2)  
*brk*(2)      OPTIONAL  
*chdir*(2)  
*chmod*(2)  
*chown*(2)  
*chroot*(2)    OPTIONAL  
*close*(2)  
*creat*(2)  
*dup*(2)  
*exec*(2)  
*exit*(2)  
*fcntl*(2)  
*fork*(2)  
*getpid*(2)  
*getuid*(2)  
*ioctl*(2)  
*kill*(2)  
*link*(2)  
*lseek*(2)  
*mknod*(2)  
*mount*(2)  
*nice*(2)      OPTIONAL  
*open*(2)  
*pause*(2)  
*pipe*(2)  
*plock*(2)    OPTIONAL  
*profil*(2)   OPTIONAL  
*ptrace*(2)   OPTIONAL  
*read*(2)  
*setpgrp*(2)  
*setuid*(2)  
*signal*(2)  
*stat*(2)  
*stime*(2)



## Contents

*sync*(2)  
*time*(2)  
*times*(2)  
*ulimit*(2)  
*umask*(2)  
*umount*(2)  
*uname*(2)  
*unlink*(2)  
*ustat*(2)  
*utime*(2)  
*wait*(2)  
*write*(2)

### Chapter 3 SUBROUTINES AND LIBRARIES

*abort*(3C)  
*abs*(3C)  
*assert*(3X)  
*bessel*(3M) OPTIONAL  
*bsearch*(3C)  
*clock*(3C)  
*conv*(3C) NLS  
*crypt*(3C)  
*ctermid*(3S)  
*ctime*(3C) NLS  
*ctype*(3C) NLS  
*cuserid*(3S)  
*directory*(3X)  
*drand48*(3C)  
*ecvt*(3C) NLS  
*end*(3C) OPTIONAL  
*erf*(3M) OPTIONAL  
*exp*(3M) OPTIONAL  
*fclose*(3S)  
*ferror*(3S)  
*floor*(3M) OPTIONAL  
*fopen*(3S)  
*fread*(3S)  
*frexp*(3C)  
*fseek*(3S)  
*ftw*(3C)  
*gamma*(3M) OPTIONAL  
*getc*(3S)  
*getcwd*(3C)  
*getenv*(3C)  
*getgrent*(3C)  
*getlogin*(3C)

<i>getopt</i> (3C)	
<i>getpass</i> (3C)	
<i>getpw</i> (3C)	
<i>getpwent</i> (3C)	
<i>gets</i> (3S)	
<i>getut</i> (3C)	
<i>hsearch</i> (3C)	
<i>hypot</i> (3M)	OPTIONAL
<i>l3tol</i> (3C)	
<i>lockf</i> (3C)	
<i>logname</i> (3X)	
<i>lsearch</i> (3C)	
<i>malloc</i> (3X)	
<i>matherr</i> (3M)	OPTIONAL
<i>memory</i> (3C)	
<i>mktemp</i> (3C)	
<i>monitor</i> (3C)	OPTIONAL
<i>perror</i> (3C)	
<i>popen</i> (3S)	
<i>printf</i> (3S)	NLS
<i>putc</i> (3S)	
<i>putenv</i> (3C)	
<i>putpwent</i> (3C)	
<i>puts</i> (3S)	
<i>qsort</i> (3C)	
<i>rand</i> (3C)	
<i>regexp</i> (3X)	
<i>scanf</i> (3S)	NLS
<i>setbuf</i> (3C)	
<i>setjmp</i> (3C)	
<i>sinh</i> (3M)	OPTIONAL
<i>sleep</i> (3C)	
<i>ssignal</i> (3C)	
<i>stdio</i> (3S)	
<i>string</i> (3C)	NLS
<i>strtod</i> (3C)	NLS
<i>strtol</i> (3C)	
<i>swab</i> (3C)	
<i>system</i> (3S)	
<i>tmpfile</i> (3S)	
<i>tmpnam</i> (3S)	
<i>trig</i> (3M)	OPTIONAL
<i>tsearch</i> (3C)	
<i>ttyname</i> (3C)	
<i>ttyslot</i> (3C)	
<i>ungetc</i> (3S)	
<i>vprintf</i> (3S)	



## Contents

### Chapter 4 FILE FORMATS

*acct*(4)  
*cpio*(4)  
*group*(4)  
*passwd*(4)  
*utmp*(4)

### Chapter 5 HEADER FILES

*acct*(5)  
*assert*(5)  
*ctype*(5)  
*dirent*(5)  
*environ*(5)  
*errno*(5)  
*fcntl*(5)  
*ftw*(5)  
*grp*(5)  
*limits*(5)  
*lock*(5)  
*malloc*(5)  
*math*(5)  
*memory*(5)  
*mon*(5)  
*pwd*(5)  
*search*(5)  
*setjmp*(5)  
*signal*(5)  
*stat*(5)  
*stdio*(5)  
*string*(5)  
*termio*(5)  
*time*(5)  
*times*(5)  
*types*(5)  
*unistd*(5)  
*ustat*(5)  
*utmp*(5)  
*utsname*(5)  
*values*(5)  
*varargs*(5)

### Chapter 6 RESERVED FOR FUTURE USE

Chapter 7 SPECIAL FILES

*console(7)*

*null(7)*

*sct(7)* OPTIONAL

*termio(7)*

*tty(7)*



# **Introduction**

## **1.1 OVERVIEW**

### **1.1.1 Rationale**

This part of the X/OPEN Portability Guide contains the System Calls and Libraries Definition, part of the X/OPEN System V Specification (XVS). It defines the system interfaces offered to application programs and the run-time behaviour of those interfaces, without imposing any particular restrictions on the way in which the interfaces are implemented.

The interfaces are defined in terms of the source code interfaces for the C programming language, which is defined in "C LANGUAGE". It is possible that some implementations may make the interfaces available to languages other than C, but this Guide does not currently define the source code interfaces for any other language.

This Specification allows an application to be built using a basic set of services that are consistent across all X/OPEN systems. Applications written in C using only these interfaces and avoiding machine dependent constructs will be portable to all X/OPEN systems.

The interfaces defined have been separated into two categories; "System Calls" and "Subroutines". This is in accordance with common practice, and should not be taken to imply that the implementation of these interfaces follows the same division.

### **1.1.2 Contents**

In accordance with common practice, the definitions of the various interfaces have been separated into seven chapters. Chapters two to five inclusive, and chapter seven, define the application interfaces.

The chapters have the following contents:

- **Chapter 1** introduces this part of the Guide and includes important notes and caveats relating to the rest of the volume.
- **Chapter 2** (System Calls) defines interfaces which are conventionally implemented as entries to the system kernel.
- **Chapter 3** (Subroutines) defines interfaces which are conventionally implemented as subroutines.
- **Chapter 4** (File Formats) defines the formats of data files which are used by system calls and subroutines.

- **Chapter 5** (Header Files) defines the contents of header files which declare constants, macros and data structures that are needed by programs using the services provided by Chapters 2 and 3.
- **Chapter 6** is empty.
- **Chapter 7** (Special Files) describes the input/output devices always present on X/OPEN systems.



## 1.2 STATUS OF INTERFACES

### 1.2.1 Mandatory

The majority of the interfaces are mandatory; they must be present in all X/OPEN systems and they must conform to the published definition.

### 1.2.2 Optional

A small number of the interfaces are optional. The presence of these interfaces is not mandatory, although if they are present they must conform to the definition. The list below shows the optional interfaces in the form *name entry*(chapter), where *name* is the name of the interface and *entry* and *chapter* are the name and chapter number of the entry in which the interface is described.

Optional Interfaces	
<i>acct</i>	<i>acct</i> (2)†
<i>brk</i>	<i>brk</i> (2)
<i>chroot</i>	<i>chroot</i> (2)†
<i>end</i>	<i>end</i> (3C)
<i>monitor</i>	<i>monitor</i> (3C)
<i>nice</i>	<i>nice</i> (2)†
<i>plock</i>	<i>plock</i> (2)†
<i>profil</i>	<i>profil</i> (2)†
<i>ptrace</i>	<i>ptrace</i> (2)†
<i>sbrk</i>	<i>brk</i> (2)
<i>sct</i>	<i>sct</i> (7)

The interfaces marked with a dagger (†) form part of the kernel extension set in the AT&T System V Interface Definition (see section **Relationship to SVID**).

The following interfaces conventionally form the standard mathematical library (3M). These routines are collectively optional, i.e., if one is present, the whole math library must be present.

Optional Math Interface	
<i>bessel</i> (3M)	<i>exp</i> (3M)
<i>erf</i> (3M)	<i>floor</i> (3M)
<i>gamma</i> (3M)	<i>hypot</i> (3M)
<i>matherr</i> (3M)	<i>sinh</i> (3M)
<i>trig</i> (3M)	

Some of the mandatory interfaces are affected by the presence or absence of the services provided by the optional interfaces. The fundamental behaviour of these interfaces is not changed; the effects are identified in the relevant interface descriptions. These interfaces are given in the following table.

Interfaces affected by options	
<i>exec</i> (2)	<i>exit</i> (2)
<i>fork</i> (2)	

1.2.3 Internationalisation

Issue 2 of the Portability Guide has introduced a set of facilities to support the development of applications capable of interacting with their users in different languages, and with appropriate cultural conventions. These are defined in "XVS INTERNATIONALISATION".

Several of these facilities take the form of enhanced versions of the standard library routines defined in the following sections of "XVS SYSTEM CALLS AND LIBRARIES". Application writers wishing to take advantage of the Internationalisation facilities should use the definitions given in "XVS INTERNATIONALISATION" in place of those given here, and to draw attention to this the affected calls have been identified on the definition page by a green "NLS".

At this issue of the Portability Guide the Internationalisation facilities have been defined together in one place so that they can be seen as a consistent and comprehensive set, and because they will be introduced into actual X/OPEN systems in a phased manner. It is likely that the definitions will be absorbed into "XVS SYSTEM CALLS AND LIBRARIES" at some future time.

The interfaces affected are summarised in the following table:

Interfaces affected by Internationalisation	
<i>conv</i> (3C)	<i>printf</i> (3C)
<i>ctime</i> (3C)	<i>scanf</i> (3C)
<i>ctype</i> (3C)	<i>string</i> (3C)
<i>ecvt</i> (3C)	<i>strtod</i> (3C)

1.2.4 Relationship to SVID

The System V Interface Definition (SVID), published by AT&T, is intended for use as a standard by applications developers. The XVS is not a distinct standard but a definition of the System V interfaces supported by X/OPEN systems, based on the SVID.

Issue 1 of the SVID was published in Spring 1985. Issue 2 published in early 1986 corrected errors and improved definitions.



Unless explicitly stated, definitions of System Calls and libraries relate to Issue 1 of the SVID. Where functional differences in Issue 2 of the SVID have been reflected, this is clearly stated in the CHANGE HISTORY.

With the exception of the mathematical routines (3M) "XVS SYSTEM CALLS AND LIBRARIES" contains as mandatory all of the SVID base interfaces.

All the interfaces within the SVID kernel extension set except those relating to shared memory, semaphores and message passing, are included in "XVS SYSTEM CALLS AND LIBRARIES" as individually optional routines.

"XVS INTER-PROCESS COMMUNICATION" defines interfaces to shared memory, semaphores and message passing, included as an interim mechanism to satisfy the immediate requirements for Inter-Process Communication facilities.

Additionally, "XVS SYSTEM CALLS AND LIBRARIES" includes a small number of interfaces taken from System V Release 2.0 but not defined in the SVID.

Wherever a definition differs from the corresponding one in the SVID, the differences are marked in the description. The rationale for such differences from the SVID is:

- Some of the SVID "FUTURE DIRECTIONS" have been included.
- The use of symbolic constants to replace explicit constants has been increased.
- Alternative wording has been used for clarification.

Where symbolic constants are used in place of explicit constants in the SVID, the symbolic constants will have the values of the SVID explicit constants.

The symbolic constants should be used wherever possible for two reasons:

- They improve readability of programs
- They protect programs from the problems which arise if the values of the explicit constants ever change

Programs written to the X/OPEN specification can easily be moved to systems that do not provide definitions for these symbolic constants. All that is required is the provision of a small number of header files containing the necessary definitions.

### 1.2.5 Subject to Change

The SVID identifies certain interfaces as possibly subject to withdrawal, by referring to them as "level 2" interfaces. The table below shows these interfaces indicating their first possible date of withdrawal:

Interfaces subject to change	
	Service Valid Until
<i>perror</i>	Dec. 31st. 1987
<i>errno</i>	"
<i>sys_errlist</i>	"
<i>sys_nerr</i>	"
<i>gsignal</i>	Nov. 30th. 1988
<i>ssignal</i>	"

The X/OPEN commitment to support of these interfaces matches that of the SVID.



## 1.3

## FORMAT OF ENTRIES

The entries in each chapter are based on a common format, not all of whose parts always appear.

The **NAME** part gives the name(s) of the entry and briefly states its purpose.

The **SYNOPSIS** part summarises the use of the entry being described. If it is necessary to include a header file to use this interface, the names of such files will be shown, e.g., `#include <stdio.h>`.

The **DESCRIPTION** part discusses the subject at hand.

The **EXAMPLE(S)** part gives example(s) of usage, where appropriate.

The **FILES** part gives the file names that are built into the subject at hand.

The **ERRORS** part gives the symbolic names of the values returned in the global variable *errno* if an error occurs.

The **RETURN VALUE** part indicates the return value, if any.

The **SEE ALSO** part gives pointers to related information.

The **APPLICATION USAGE** part gives information about the way that the subject at hand should be used.

The **FUTURE DIRECTIONS** part is generally copied from the SVID, unless the change indicated in the SVID has already been adopted. Comments found in this section should be used as a guide to current thinking; there is not necessarily a commitment to implement all of these future directions in their entirety.

The **CHANGE HISTORY** part shows the derivation of the entry and any changes which have been made to it.

The formal description consists only of the **NAME**, **SYNOPSIS**, **DESCRIPTION**, **FILES**, **ERRORS** and **RETURN VALUES** parts.

The following typographical conventions are used throughout this part:

**Boldface** strings are literals and are to be typed just as they appear.

*Italic* strings usually represent substitutable argument prototypes and the names of entries found elsewhere.

Names in upper case surrounded by braces, e.g. `{CONST}` represent constants which may be declared in appropriate header files by means of the C `#define` facility. For portability, only the symbolic names should be used, never the value that a particular implementation may happen to use. The values of most of these constants are defined in `<limits.h>`, `<values.h>` or `<unistd.h>`.

Square brackets `[]` around an argument prototype indicate that the argument is optional. When an argument prototype is given as *name* or *file*, it always refers to a *file* name.

The notation `<file.h>` indicates a header file, also known as an include file, which is supplied as part of the applications development system, see Chapter 5 and FILE INCLUSION in "C LANGUAGE".

Ellipses ... are used to show that the previous argument may be repeated.

Whenever referring to a subject described in chapters 2 - 7, its chapter number is appended to its name, in parentheses. For example: `access(2)`.

If a chapter number of 1 appears, this refers to a definition included in "XVS COMMANDS AND UTILITIES". This convention is in line with standard practice.



## 1.4 DEFINITIONS

Many special terms are used in the interface definitions. The descriptions of these terms follows.

### 1.4.1 Character and Block Special Files

Character and block special files are used to refer to physical devices. Certain restrictions may apply to use of character and block special files which are implementation dependent.

### 1.4.2 Character Sets

Many of the services refer to the ASCII or ISO 646 character set.

For a full definition refer to "XVS INTERNATIONALISATION".

### 1.4.3 Command Interpreter

It is possible for applications to invoke utilities through a number of interfaces, which are collectively considered to act as command interpreters. The most obvious of these are *sh*(1) and *system*(3S), although *popen*(3S) and the various forms of *exec*(2) may also be considered to behave as interpreters.

### 1.4.4 Directory

Directories organise files into a hierarchical system where directories are the nodes of the hierarchy. A directory is a file that catalogues the list of files, including directories (sub-directories), that are directly beneath it in the hierarchy. Entries in a directory file are called links, which associate a file identifier with a file name. By convention, a directory contains at least two links, *.* and *..*, referred to as *dot* and *dot-dot* respectively. *Dot* refers to the directory itself and *dot-dot* refers to its parent directory. The root directory, which is the top-most node of the hierarchy, has itself as its parent directory. The path-name of the root directory is */* and the parent directory of the root directory is */*.

### 1.4.5 Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group-ID bit set, see *exec*(2). In addition, they can be changed with *setuid*(2) and *setgid*(2), respectively.

### 1.4.6 Fifo Special Files

A fifo special file is a named "pipe", see *mknod*(1), *pipe*(2) and *mknod*(2). Normally, a fifo special file is opened in conjunction by two or more separate processes. One or more processes write data to the fifo special file and another process reads this same data from the file on a "first-in-first-out" basis. Seeks on a fifo special file have no meaning and cause the [ESPIPE] error.

#### 1.4.7 File Access Permissions

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user, and in the case of execute permission, at least one of the execute bits is set.

The effective user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (S\_IRWXU) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion (S\_IRWXG) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion (S\_IRWXO) of the file mode is set.

Otherwise, the corresponding permissions are denied.

#### 1.4.8 File Descriptor

A file descriptor is a small integer used to identify a file for the purpose of doing I/O. The value of a file descriptor is from 0 to {OPEN\_MAX}-1. A process may have no more than {OPEN\_MAX} file descriptors open simultaneously.

A file descriptor has associated with it information used in performing I/O on the file: a file pointer that marks the next position within the file where I/O will begin; file status and access modes (e.g. read, write, read/write), see *open(2)*; and close-on-exec flag, see *fcntl(2)*. Multiple file descriptors may identify the same file. A file descriptor is returned by system routines such as *creat(2)*, *dup(2)*, *fcntl(2)*, *open(2)*, or *pipe(2)*. The file descriptor is used as an argument by routines such as *read(2)*, *write(2)*, *ioctl(2)* and *close(2)*.

#### 1.4.9 File Name

Names consisting of 1 to {NAME\_MAX} characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding the characters "null" and "slash".

**Note:** it is generally unwise to use \*, ?, !, [, or ] as part of file names because of the special meaning attached to these characters for filename expansion by some command interpreters, see *system(3S)*. Other characters to avoid are the hyphen, blank, tab, <, >, backslash, single and double quotes, accent grave, vertical bar, carat, curly braces and parentheses. It is also advisable to avoid the use of non-printing characters in file names.



**1.4.10 Parent Process ID**

A new process is created by a currently active process, see *fork(2)*. The parent process ID of a process is the process ID of its creator, unless the creator has already exited, see *exit(2)*.

**1.4.11 Path Name and Path Prefix**

In a C program a path name is a null-terminated character-string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name. The null string is undefined and may be considered an error.

More precisely, a path name is a null-terminated character string constructed as follows:

```
<path-name>::=<file-name>|<path-prefix><file-name>|/|.|..
<path-prefix>::=<rtprefix>|/<rtprefix>|/
<rtprefix>::=<dirname>/|<rtprefix><dirname>/
```

where <file-name> is a string of 1 to {NAME\_MAX} characters other than slash and null, and <dirname> is a string of 1 to {NAME\_MAX} characters (other than slash and null) that names a directory.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory. The meanings of . and .. are defined under *Directory*.

The result of names not produced by the grammar is undefined.

**1.4.12 Process Group ID**

Each active process is a member of a process group. The process group is uniquely identified by a non-negative integer, called the process group ID, which is the process ID of the group leader (see below). This grouping permits the signaling of related processes, see *kill(2)*. This grouping is also used to terminate a group of related processes upon termination of the group leader, see *exit(2)* and *signal(2)*.

**1.4.13 Process Group Leader**

A process group leader is any process whose process group ID is the same as its process ID. Any process may detach itself from its current process group and become a process group leader by calling *setpgid(2)*. A process inherits the process group ID of the process that created it, see *fork(2)* and *exec(2)*.

**1.4.14 Process ID**

Each active process in the system is uniquely identified by a non-negative integer called a process ID. The range of this ID is from 0 to {PID\_MAX}. Process IDs between 0 and {SYSPID\_MAX} inclusive are reserved for special system processes.

#### 1.4.15 Real User ID and Real Group ID

Each user allowed on the system is identified by a non-negative integer called a real user ID.

Each user is also a member of a group. The group is identified by a non-negative integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process. They can be reset with *setuid(2)* and *setgid(2)*, respectively.

#### 1.4.16 Root Directory and Current Working Directory

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. The root directory of a process need not be the root directory of the root file system, see *chroot(2)*.

#### 1.4.17 Special Processes

Special processes are system processes as, for example, a system's process scheduler. Process IDs between 0 and {SYSPID\_MAX} inclusive are reserved for special system processes.

#### 1.4.18 Super-user

A process is recognised as a *super-user* process and is granted special privileges if its effective user ID is 0.

#### 1.4.19 Terminal Group

A terminal group is a group of processes associated with a particular terminal. The group is used to select those processes which will receive signals generated by the terminal handling part of a system. In particular, *SIGHUP*, *SIGINT* and *SIGQUIT* are commonly caused in this way. On asynchronous lines (see *termio(7)*) the latter two are generated on receipt of the *INTR* and *QUIT* characters, if the appropriate mode has been enabled. This grouping is also used to terminate a group of related processes upon termination of the group leader, see *exit(2)* and *signal(2)*.

#### 1.4.20 Tty Group ID

The tty group ID is a non-negative integer used to identify a terminal group.



## 1.5 SIGNALS

To be portable, applications should only send, catch or ignore the following signals:

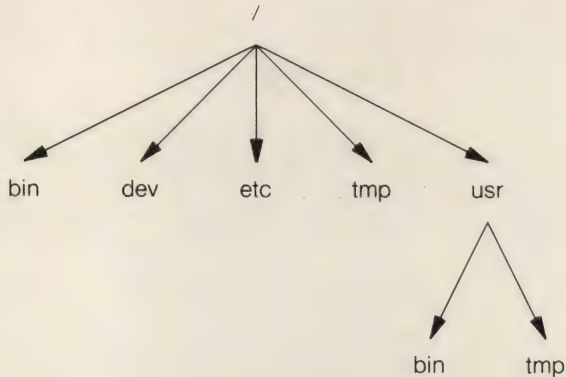
Signal	Description
SIGHUP	hangup
SIGINT	interrupt (rubout)
SIGQUIT	quit
SIGILL	illegal instruction (not reset when caught)
SIGTRAP	trace trap (not reset when caught)
SIGABRT†	process abort signal
SIGFPE	floating point exception
SIGKILL	kill (cannot be caught or ignored)
SIGSYS	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal from kill
SIGUSR1	user defined signal 1
SIGUSR2	user defined signal 2

The signal marked above SIGABRT† has been included from a FUTURE DIRECTION indicated in the SVID.

It should be noted that in some cases signals can be sent to processes as a result of keystrokes on terminals. See the definition of *Terminal Group*.

## 1.6 DIRECTORY STRUCTURE

Below is a diagram of the minimal directory tree structure present on any system.



The following guidelines apply to the contents of these directories:

**/bin**, **/dev**, **/etc** and **/tmp** are primarily for the use of the system. Most applications should never create files in any of these directories, though they may read and execute them.

**/bin** contains executable system commands (utilities), although it may be empty.

**/dev** contains special files (I/O devices). Those which are always present in X/OPEN systems are defined in Chapter 7.

**/etc** contains system data files such as **/etc/passwd**. It may also contain some executable files which are used by the system; these are not intended to be accessible to the ordinary user.

**/tmp** contains temporary files created by any utilities in **/bin**, and other system processes. Applications should use **/usr/tmp**.

**/usr/bin** and **/usr/tmp** can be used by applications as well as the system.

**/usr/bin** contains (user-level) executable application commands and system commands.

**/usr/tmp** contains temporary files created by application programs and by the system.

If the system is re-started after being halted for any reason, applications cannot rely on the contents of **/tmp** or **/usr/tmp** remaining undisturbed. It is common for both directories to be emptied by the restart procedures.



## 1.7 ENVIRONMENTAL VARIABLES

An array of strings is made available by *exec(2)* when a process begins execution, see also *system(3S)*. These strings are described more fully in *environ(5)*, but the minimum set of strings that can be expected to exist and be set in any X/OPEN environment are:

<i>Variable</i>	<i>Use</i>
HOME	Full pathname of the user's home directory, set when the user signs on.
LOGNAME	Login name.
PATH	A colon separated ordered list of pathnames that determine the search sequence used in locating files.
TERM	The kind of terminal for which output is prepared.
TZ	Time zone information. See also <i>ctime(3C)</i> .

The X/OPEN Internationalisation Definition introduces further environmental variables. These are fully defined in "XVS INTERNATIONALISATION".

## 1.8 SYSTEM RESIDENT DATA FILES

The only system-resident data files implied by the X/OPEN system definition are given in the following table, together with a reference to the appropriate chapter and entry to find their definitions.

data file	reference
/etc/group	<i>group</i> (4)
/etc/passwd	<i>passwd</i> (4)
/etc/utmp	<i>utmp</i> (4)
/etc/profile	<i>see below</i>
/etc/wtmp	<i>utmp</i> (4)

*/etc/profile* is a file of *sh*(1) commands which is used to establish an environment for users when they initially sign-on to an X/OPEN system. The mechanism for the signing-on procedure is not defined, nor is it necessarily the case that users will use *sh*(1) as their command interpreter. However, the file will contain an assignment to the **PATH** environment variable. This is a fundamental requirement to enable the command interpreter in use to locate the utilities in the file system.

## 1.9 SPECIAL FILES

The names of specific I/O devices are known as "special files". The only ones present in every X/OPEN system are given below, together with reference to their descriptions. The ones marked (†) are only present in systems supporting the source code transfer standard and need *not* be present in every system.

device	reference
/dev/console	<i>console</i> (7)
/dev/null	<i>null</i> (7)
/dev/tty	<i>tty</i> (7)
/dev/sctfdm	<i>sct</i> (7)†
/dev/sctmtm	<i>sct</i> (7)†
/dev/rsctfdm	<i>sct</i> (7)†
/dev/rsctmtm	<i>sct</i> (7)†



## 1.10 CAVEATS

### 1.10.1 Null Pointers

The descriptions of some functions refer to the NULL pointer. This is the value that is obtained by casting 0 into a pointer i.e., *(char \*) 0*. The C language guarantees that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

For consistency with the C language definition, this interpretation of the NULL pointer has been retained. However, reference should be made to the notes on C program portability in "C LANGUAGE", where it is indicated that some systems do not support this definition of the NULL pointer.

(A NULL pointer should not be confused with the NULL character. The NULL character is a character with the value 0, represented in the C language as '\0'. A string, or NULL-terminated character array, is a sequence of characters, the last of which is the NULL character. A NULL string is an array of characters which contains only the NULL character.)

### 1.10.2 Termio(7)

The SVID interface for locally connected asynchronous lines, *termio(7)*, is a mandatory part of the SVID base. Some X/OPEN systems may support synchronous lines, or may support asynchronous lines over networks. In both cases, it is impossible to support the full definition of *termio(7)*. For these reasons X/OPEN cannot guarantee full support for *termio(7)* in all cases. This also slightly affects the functionality of *read(2)* and *open(2)*, with respect to the *O\_NDELAY* flag.

The *open(2)* description discusses what happens while waiting for a carrier to be detected on a communication line. It should be noted that even if full support is otherwise provided for *termio(7)*, the hardware driving the line may not respond to the modem control signals. In this case, it will appear as if carrier is permanently present and there will be no delay when opening such a line.

### 1.10.3 Process IDs

The values of a Process ID are specified to range from zero to {PID\_MAX}. On many systems, the value of {PID\_MAX} is small enough to imply that variables of type **short** provide adequate precision to store such a value. This is not always a justifiable assumption, and application developers are warned that **int** variables should be used for this purpose.

The *ut\_pid* field of the *utmp* structure (see *utmp(5)*) is explicitly declared to be of type **short**, in line with the SVID. On some systems its type may be different; programming practices which rely on the type of *ut\_pid* should be avoided. In particular, if it is necessary to take the address of this structure member, the type of the resulting pointer will depend on the type of *ut\_pid*.

#### 1.10.4 The files `<values.h>` and `<limits.h>`

A number of limits and values which are of importance to system developers are system dependent. Their values are available in the include files `<limits.h>` and `<values.h>`, as symbolic constants. These constants are referred to in many places in the interface definitions; for example `{SYSPID_MAX}`. The file `<values.h>` is part of the SVID base. The file `<limits.h>`, part of the `/usr/group` standard, defines additional values and has also been included in the XVS.

In some cases, the same value is defined in both of these files, although the name used to refer to the value differs. In X/OPEN systems, both names are set to the same value.

Applications developers may choose to include either, neither or both of these files in a particular application, depending upon their needs. The interfaces defined in the following chapters can always be used without it being necessary to include either of these files.



### 1.11 ERRORS AND EXCEPTIONS

Most system calls and subroutines can result in exceptions, known as "error returns". An error condition is indicated by an otherwise impossible returned value. This is almost always -1; the individual descriptions specify the details.

As with normal arguments, all return codes and values from functions are of type `int` unless otherwise noted. An error number is also made available in the external `int` variable `errno`. `Errno` is not cleared on successful calls, so it should be tested only after an error has been indicated.

A full list of error names is defined in `errno(5)`. Only these symbolic names for error numbers should be used in programs, since the actual value of the error number may vary with the implementation. Certain implementations may not return all of the error types listed. Other implementations may return errors which are not included on the list.

The [EFAULT] error is caused by a program referencing data outside its legitimate address space. The reliable detection of this error cannot be guaranteed.

Functions in the Math Library (3M) may return the conventional values 0 or HUGE (the largest single-precision floating-point number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable `errno` is set to the value [EDOM] or [ERANGE].

## 1.12 LIST OF SERVICES

A list of all services specified in the XVS follows. Some of the entries describe a number of services, so to find a particular one knowing its name, look in the column labeled "Service". The entry describing the service is given in the column on its right. For example, both *isascii* and *isspace* are to be found under the overall heading of *ctype(3C)*.

Services marked with a (†) are optional. Services marked with a (§) are affected by internationalisation. Utilities marked with a (□) are 8-bit clean in systems which conform to "XVS INTERNATIONALISATION". System calls marked with a (●) are defined in "XVS INTER-PROCESS COMMUNICATION".

The column marked SVID shows where the corresponding SVID entry is located.



Interface	Entry	SVID
abort	abort(3C)	BA_OS
abs	abs(3C)	BA_LIB
access	access(2)	BA_OS
acct	acct(2)†	KE_OS
acct	acct(4)	
acct	acct(5)	
acos	trig(3M)†	BA_LIB
admin	admin(1D)	SD_CMD
alarm	alarm(2)	BA_OS
ar	ar(1)	BU_CMD
as	as(1D)†	SD_CMD
asctime	ctime(3C)‡	BA_LIB
asin	trig(3M)†	BA_LIB
assert	assert(3X)	SD_LIB
assert	assert(5)	
at	at(1)	AU_CMD
atan	trig(3M)†	BA_LIB
atan2	trig(3M)†	BA_LIB
atof	strtod(3C)‡	BA_LIB
atoi	strtol(3C)	BA_LIB
atol	strtol(3C)	BA_LIB
awk	awk(1)	BU_CMD
banner	banner(1)	BU_CMD
basename	basename(1)	BU_CMD
batch	at(1)	AU_CMD
brk	brk(2)†	
bsearch	bsearch(3C)	BA_LIB
cal	cal(1)	BU_CMD
calendar	calendar(1)	BU_CMD
calloc	malloc(3X)	BA_OS
cancel	lp(1)	AU_CMD
cat	cat(1)□	BU_CMD
cc	cc(1D)	SD_CMD
cd	cd(1)□	BU_CMD
ceil	floor(3M)†	BA_LIB
cflow	cflow(1D)	SD_CMD
chdir	chdir(2)	BA_OS
chgrp	chown(1)	AU_CMD
chmod	chmod(1)	BU_CMD
chmod	chmod(2)	BA_OS
chown	chown(1)	AU_CMD
chown	chown(2)	BA_OS

Interface	Entry	SVID
chroot	chroot(1)	SD_CMD
chroot	chroot(2)†	KE_OS
clearerr	ferror(3S)	BA_OS
clock	clock(3C)	BA_LIB
close	close(2)	BA_OS
closedir	directory(3X)	
cmp	cmp(1)	BU_CMD
col	col(1)	BU_CMD
comm	comm(1)	BU_CMD
console	console(7)	BA_ENV
cos	trig(3M)†	BA_LIB
cosh	sinh(3M)†	BA_LIB
cp	cp(1)□	BU_CMD
cpio	cpio(1)□	BU_CMD
cpio	cpio(4)	
cpp	cpp(1D)	SD_CMD
creat	creat(2)	BA_OS
crontab	crontab(1)	AU_CMD
crypt	crypt(3C)	BA_LIB
csplit	csplit(1)	AU_CMD
ctermid	ctermid(3S)	BA_LIB
ctime	ctime(3C)‡	BA_LIB
ctype	ctype(5)	
cu	cu(1)	AU_CMD
cuserid	cuserid(3S)	
cut	cut(1)	BU_CMD
cxref	cxref(1D)	SD_CMD
date	date(1)	BU_CMD
daylight	ctime(3C)‡	BA_LIB
dd	dd(1)	AU_CMD
delta	delta(1D)	SD_CMD
df	df(1)	BU_CMD
diff	diff(1)	BU_CMD
dircmp	dircmp(1)	AU_CMD
dirent	dirent(5)	
dirname	basename(1)	BU_CMD
dis	dis(1D)†	SD_CMD
drand48	drand48(3C)	BA_LIB
du	du(1)	BU_CMD
dup	dup(2)	BA_OS
echo	echo(1)□	BU_CMD
ecvt	ecvt(3C)‡	

Interface	Entry	SVID
ed	ed(1)□	BU_CMD
edata	end(3C)†	
egrep	egrep(1)	AU_CMD
encrypt	crypt(3C)	BA_LIB
end	end(3C)†	
endgrent	getgrent(3C)	SD_LIB
endpwent	getpwent(3C)	SD_LIB
endutent	getut(3C)	SD_LIB
env	env(1D)	SD_CMD
erand48	drand48(3C)	BA_LIB
erf	erf(3M)†	BA_LIB
erfc	erf(3M)†	BA_LIB
errno	errno(5)	BA_ENV
errno	perror(3C)	BA_LIB
etext	end(3C)†	
ex	ex(1)	AU_CMD
execl	exec(2)	BA_OS
execle	exec(2)	BA_OS
execvp	exec(2)	BA_OS
execv	exec(2)	BA_OS
execve	exec(2)	BA_OS
execvp	exec(2)	BA_OS
_exit	exit(2)	BA_OS
exit	exit(2)	BA_OS
exp	exp(3M)†	BA_LIB
expr	expr(1)□	BU_CMD
fabs	floor(3M)†	BA_LIB
false	true(1)	BU_CMD
fclose	fclose(3S)	BA_OS
fcntl	fcntl(2)	BA_OS
fcntl	fcntl(5)	
fcvt	ecvt(3C)‡	
fdopen	fopen(3S)	BA_OS
feof	ferror(3S)	BA_OS
ferror	ferror(3S)	BA_OS
fflush	fclose(3S)	BA_OS
fgetc	getc(3S)	BA_LIB
fgetgrent	getgrent(3C)	SD_LIB
fgetpwent	getpwent(3C)	SD_LIB
fgets	gets(3S)	BA_LIB
fgrep	egrep(1)	AU_CMD
file	file(1)	BU_CMD

Interface	Entry	SVID
fileno	ferror(3S)	BA_OS
find	find(1)□	BU_CMD
floor	floor(3M)†	BA_LIB
fmod	floor(3M)†	BA_LIB
fopen	fopen(3S)	BA_OS
fork	fork(2)	BA_OS
fprintf	printf(3S)‡	BA_LIB
fputc	putc(3S)	BA_LIB
fputs	puts(3S)	BA_LIB
fread	fread(3S)	BA_OS
free	malloc(3X)	BA_OS
freopen	fopen(3S)	BA_OS
frexp	frexp(3C)	BA_LIB
fscanf	scanf(3S)‡	BA_LIB
fseek	fseek(3S)	BA_OS
fstat	stat(2)	BA_OS
ftell	fseek(3S)	BA_OS
ftw	ftw(3C)	BA_LIB
ftw	ftw(5)	
fwrite	fread(3S)	BA_OS
gamma	gamma(3M)†	BA_LIB
gcvt	ecvt(3C)‡	
get	get(1D)	SD_CMD
getc	getc(3S)	BA_LIB
getchar	getc(3S)	BA_LIB
getcwd	getcwd(3C)	BA_OS
getegid	getuid(2)	BA_OS
getenv	getenv(3C)	BA_LIB
geteuid	getuid(2)	BA_OS
getgid	getuid(2)	BA_OS
getgrent	getgrent(3C)	SD_LIB
getgrgid	getgrent(3C)	SD_LIB
getgrnam	getgrent(3C)	SD_LIB
getlogin	getlogin(3C)	SD_LIB
getopt	getopt(3C)	BA_LIB
getpass	getpass(3C)	SD_LIB
getpgrp	getpid(2)	BA_OS
getpid	getpid(2)	BA_OS
getppid	getpid(2)	BA_OS
getpw	getpw(3C)	
getpwent	getpwent(3C)	SD_LIB
getpwnam	getpwent(3C)	SD_LIB



Interface	Entry	SVID
getpwuid	getpwent (3C)	SD_LIB
gets	gets (3S)	BA_LIB
getuid	getuid (2)	BA_OS
getutent	getut (3C)	SD_LIB
getutid	getut (3C)	SD_LIB
getutline	getut (3C)	SD_LIB
getw	getc (3S)	BA_LIB
gmtime	ctime (3C)‡	BA_LIB
grep	grep (1)□	BU_CMD
group	group (4)	
grp	grp (5)	
gsignal	ssignal (3C)	BA_LIB
hcreate	hsearch (3C)	BA_LIB
hdestroy	hsearch (3C)	BA_LIB
hsearch	hsearch (3C)	BA_LIB
hypot	hypot (3M)†	BA_LIB
id	id (1)	AU_CMD
ioctl	ioctl (2)	BA_OS
ipc	ipc (2)●†	
ipc	ipc (5)●	
isalnum	ctype (3C)‡	BA_LIB
isalpha	ctype (3C)‡	BA_LIB
isascii	ctype (3C)‡	BA_LIB
isatty	ttyname (3C)	BA_LIB
iscntrl	ctype (3C)‡	BA_LIB
isdigit	ctype (3C)‡	BA_LIB
isgraph	ctype (3C)‡	BA_LIB
islower	ctype (3C)‡	BA_LIB
isprint	ctype (3C)‡	BA_LIB
ispunct	ctype (3C)‡	BA_LIB
isspace	ctype (3C)‡	BA_LIB
isupper	ctype (3C)‡	BA_LIB
isxdigit	ctype (3C)‡	BA_LIB
j0	bessel (3M)†	BA_LIB
j1	bessel (3M)†	BA_LIB
jn	bessel (3M)†	BA_LIB
join	join (1)	AU_CMD
rand48	drand48 (3C)	BA_LIB
kill	kill (1)	BU_CMD
kill	kill (2)	BA_OS
l3tol	l3tol (3C)	
lcong48	drand48 (3C)	BA_LIB

Interface	Entry	SVID
ld	ld (1D)	SD_CMD
ldexp	frexp (3C)	BA_LIB
lex	lex (1D)	SD_CMD
lfind	lsearch (3C)	BA_LIB
limits	limits (5)	
line	line (1)	BU_CMD
link	link (2)	BA_OS
lint	lint (1D)	SD_CMD
ln	cp (1)□	BU_CMD
localtime	ctime (3C)‡	BA_LIB
lock	lock (5)	
lockf	lockf (3C)	BA_OS
log	exp (3M)†	BA_LIB
log10	exp (3M)†	BA_LIB
logname	logname (1)	AU_CMD
logname	logname (3X)	
longjmp	setjmp (3C)	BA_LIB
lorder	lorder (1D)	SD_CMD
lp	lp (1)	AU_CMD
lpstat	lpstat (1)	AU_CMD
lrand48	drand48 (3C)	BA_LIB
ls	ls (1)□	BU_CMD
lsearch	lsearch (3C)	BA_LIB
lseek	lseek (2)	BA_OS
l3tol	l3tol (3C)	
m4	m4 (1D)	SD_CMD
mail	mail (1)	BU_CMD
mailx	mailx (1)†	AU_CMD
make	make (1D)	SD_CMD
mallinfo	malloc (3X)	BA_OS
malloc	malloc (3X)	BA_OS
malloc	malloc (5)	
mallopt	malloc (3X)	BA_OS
math	math (5)	
matherr	matherr (3M)†	BA_LIB
memccpy	memory (3C)	BA_LIB
memchr	memory (3C)	BA_LIB
memcmp	memory (3C)	BA_LIB
memcpy	memory (3C)	BA_LIB
memory	memory (5)	
memset	memory (3C)	BA_LIB
mesg	mesg (1)	AU_CMD

Interface	Entry	SVID
mkdir	mkdir(1)□	BU_CMD
mknod	mknod(1)†	AS_CMD
mknod	mknod(2)	BA_OS
mktemp	mktemp(3C)	BA_LIB
modf	frexp(3C)	BA_LIB
mon	mon(5)	
monitor	monitor(3C)†	SD_LIB
mount	mount(2)	BA_OS
rand48	drand48(3C)	BA_LIB
msg	msg(5)●	
msgctl	msgctl(2)●†	KE_OS
msgget	msgget(2)●†	KE_OS
msgrcv	msgop(2)●†	KE_OS
msgsnd	msgop(2)●†	KE_OS
mv	cp(1)□	BU_CMD
newgrp	newgrp(1)†	AU_CMD
news	news(1)†	AU_CMD
nice	nice(2)†	KE_OS
nl	nl(1)	BU_CMD
nm	nm(1D)	SD_CMD
nohup	nohup(1)	BU_CMD
rand48	drand48(3C)	BA_LIB
null	null(7)	BA_ENV
od	od(1)	AU_CMD
open	open(2)	BA_OS
opendir	directory(3X)	
pack	pack(1)	BU_CMD
passwd	passwd(1)	AU_CMD
passwd	passwd(4)	
paste	paste(1)	BU_CMD
pause	pause(2)	BA_OS
pcat	pack(1)	BU_CMD
pclose	popen(3S)	BA_OS
perror	perror(3C)	BA_LIB
pg	pg(1)	BU_CMD
pipe	pipe(2)	BA_OS
plock	plock(2)†	KE_OS
popen	popen(3S)	BA_OS
pow	exp(3M)†	BA_LIB
pr	pr(1)□	BU_CMD
printf	printf(3S)‡	BA_LIB
prof	prof(1D)†	SD_CMD

Interface	Entry	SVID
profil	profil(2)†	KE_OS
prs	prs(1D)	SD_CMD
ps	ps(1)	BU_CMD
ptrace	ptrace(2)†	KE_OS
putc	putc(3S)	BA_LIB
putchar	putc(3S)	BA_LIB
putenv	putenv(3C)	BA_LIB
putpwent	putpwent(3C)	SD_LIB
puts	puts(3S)	BA_LIB
pututline	getut(3C)	SD_LIB
putw	putc(3S)	BA_LIB
pwd	pwd(1)	BU_CMD
pwd	pwd(5)	BA_ENV
qsort	qsort(3C)	BA_LIB
rand	rand(3C)	BA_LIB
read	read(2)	BA_OS
readdir	directory(3X)	
realloc	malloc(3X)	BA_OS
red	ed(1)□	BU_CMD
regexp	regexp(3X)	BA_LIB
rewind	fseek(3S)	BA_OS
rewinddir	directory(3X)	
rm	rm(1)□	BU_CMD
rmdel	rmdel(1D)	SD_CMD
rmdir	rm(1)□	BU_CMD
HOME	environ(5)	BA_ENV
LOGNAME	environ(5)	BA_ENV
PATH	environ(5)	BA_ENV
TERM	environ(5)	BA_ENV
TZ	environ(5)	BA_ENV
sact	sact(1D)	SD_CMD
sbrk	brk(2)†	
scanf	scanf(3S)‡	BA_LIB
sctfd	sct(7)†	
sctmt	sct(7)†	
sdb	sdb(1D)†	SD_CMD
search	search(5)	
sed	sed(1)□	BU_CMD
seed48	drand48(3C)	BA_LIB
seekdir	directory(3X)	
sem	sem(5)●	
semctl	semctl(2)●†	KE_OS



Interface	Entry	SVID
semget	semget (2)●†	KE_OS
semop	semop (2)●†	KE_OS
setbuf	setbuf (3C)	BA_LIB
setgid	setuid (2)	BA_OS
setgrent	getgrent (3C)	SD_LIB
setjmp	setjmp (3C)	BA_LIB
setjmp	setjmp (5)	
setkey	crypt (3C)	BA_LIB
setpgrp	setpgrp (2)	BA_OS
setpwent	getpwent (3C)	SD_LIB
setuid	setuid (2)	BA_OS
setutent	getut (3C)	SD_LIB
setvbuf	setbuf (3C)	BA_LIB
sh	sh (1)□	BU_CMD
shl	shl (1)†	AU_CMD
shm	shm (5)●	
shmat	shmop (2)●†	KE_OS
shmctl	shmctl (2)●†	KE_OS
shmctl	shmop (2)●†	KE_OS
shmget	shmget (2)●†	KE_OS
signal	signal (2)	BA_OS
signal	signal (5)	
signgam	gamma (3M)†	BA_LIB
sin	trig (3M)†	BA_LIB
sinh	sinh (3M)†	BA_LIB
size	size (1D)	SD_CMD
sleep	sleep (1)	BU_CMD
sleep	sleep (3C)	BA_OS
sort	sort (1)□	BU_CMD
spell	spell (1)	BU_CMD
split	split (1)	BU_CMD
sprintf	printf (3S)‡	BA_LIB
sqrt	exp (3M)†	BA_LIB
srand	rand (3C)	BA_LIB
srand48	drand48 (3C)	BA_LIB
sscanf	scanf (3S)‡	BA_LIB
ssignal	ssignal (3C)	BA_LIB
stat	stat (2)	BA_OS
stat	stat (5)	
stdio	stdio (3S)	
stdio	stdio (5)	
stime	stime (2)	BA_OS

Interface	Entry	SVID
strcat	string (3C)‡	BA_LIB
strchr	string (3C)‡	BA_LIB
strcmp	string (3C)‡	BA_LIB
strcpy	string (3C)‡	BA_LIB
strcsn	string (3C)‡	BA_LIB
string	string (5)	
strip	strip (1D)	SD_CMD
strlen	string (3C)‡	BA_LIB
strncat	string (3C)‡	BA_LIB
strncmp	string (3C)‡	BA_LIB
strncpy	string (3C)‡	BA_LIB
strpbrk	string (3C)‡	BA_LIB
strchr	string (3C)‡	BA_LIB
strspn	string (3C)‡	BA_LIB
strtod	strtod (3C)‡	BA_LIB
strtok	string (3C)‡	BA_LIB
strtol	strtol (3C)	BA_LIB
stty	stty (1)	AU_CMD
su	su (1)	AU_CMD
sum	sum (1)	BU_CMD
swab	swab (3C)	BA_LIB
sync	sync (2)	BA_OS
sys_errlist	perror (3C)	BA_LIB
sys_nerr	perror (3C)	BA_LIB
system	system (3S)	BA_OS
tabs	tabs (1)	AU_CMD
tail	tail (1)	BU_CMD
tan	trig (3M)†	BA_LIB
tanh	sinh (3M)†	BA_LIB
tar	tar (1)	AU_CMD
tdelete	tsearch (3C)	BA_LIB
tee	tee (1)	BU_CMD
telldir	directory (3X)	
tempnam	tmpnam (3S)	BA_LIB
termio	termio (5)	
termio	termio (7)	BA_ENV
test	test (1)	BU_CMD
tfind	tsearch (3C)	BA_LIB
time	time (1D)	SD_CMD
time	time (2)	BA_OS
time	time (5)	
times	times (2)	BA_OS

Interface	Entry	SVID
times	times (5)	
timezone	ctime (3C)‡	BA_LIB
tmpfile	tmpfile (3S)	BA_LIB
tmpnam	tmpnam (3S)	BA_LIB
toascii	conv (3C)‡	BA_LIB
_tolower	conv (3C)‡	BA_LIB
tolower	conv (3C)‡	BA_LIB
touch	touch (1)	BU_CMD
_toupper	conv (3C)‡	BA_LIB
toupper	conv (3C)‡	BA_LIB
tr	tr (1)□	BU_CMD
true	true (1)	BU_CMD
tsearch	tsearch (3C)	BA_LIB
tsort	tsort (1D)	SD_CMD
tty	tty (1)	AU_CMD
tty	tty (7)	BA_ENV
ttyname	ttyname (3C)	BA_LIB
ttyslot	ttyslot (3C)	
twalk	tsearch (3C)	BA_LIB
types	types (5)	
tzname	ctime (3C)‡	BA_LIB
tzset	ctime (3C)‡	BA_LIB
ulimit	ulimit (2)	BA_OS
umask	umask (1)	BU_CMD
umask	umask (2)	BA_OS
umount	umount (2)	BA_OS
uname	uname (1)	BU_CMD
uname	uname (2)	BA_OS
unget	unget (1D)	SD_CMD
ungetc	ungetc (3S)	BA_LIB
uniq	uniq (1)	BU_CMD
unistd	unistd (5)	
unlink	unlink (2)	BA_OS
unpack	pack (1)	BU_CMD
ustat	ustat (2)	BA_OS

Interface	Entry	SVID
ustat	ustat (5)	
utime	utime (2)	BA_OS
utmp	utmp (4)	
utmp	utmp (5)	
utmpname	getut (3C)	SD_LIB
utsname	utsname (5)	
uucp	uucp (1)	AU_CMD
uulog	uucp (1)	AU_CMD
uuname	uucp (1)	AU_CMD
uupick	uuto (1)	AU_CMD
uustat	uustat (1)	AU_CMD
uuto	uuto (1)	AU_CMD
uux	uux (1)	AU_CMD
val	val (1D)	SD_CMD
values	values (5)	
varargs	varargs (5)	
vprintf	vprintf (3S)	BA_LIB
vi	vi (1)	AU_CMD
vprintf	vprintf (3S)	BA_LIB
vsprintf	vprintf (3S)	BA_LIB
wait	wait (1)	BU_CMD
wait	wait (2)	BA_OS
wall	wall (1)	AU_CMD
wc	wc (1)□	BU_CMD
what	what (1D)	SD_CMD
who	who (1)	AU_CMD
write	write (1)	AU_CMD
write	write (2)	BA_OS
wtmp	utmp (4)	
xargs	xargs (1D)	SD_CMD
y0	bessel (3M)†	BA_LIB
y1	bessel (3M)†	BA_LIB
yacc	yacc (1D)	SD_CMD
yn	bessel (3M)†	BA_LIB



## System Calls

This chapter describes system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always -1; the individual descriptions specify the details.

As with normal arguments, all return codes and values from functions are of type `int` unless otherwise noted. An error number is also made available in the external `int` variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

The error names are described in *errno*(5).





## NAME

access — determine accessibility of a file

## SYNOPSIS

```
#include <unistd.h>
```

```
int access (path, amode)
char *path;
int amode;
```

## DESCRIPTION

*Path* points to a path name naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID or equivalent in place of the effective group ID. The value of *amode* is the sum of the access modes to be checked as defined in <unistd.h>:

R_OK	04	read
W_OK	02	write
X_OK	01	execute (search)
F_OK	00	check existence of file

Thus, the value of *amode* should be the sum of the values of the access modes to be checked.

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits. Members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

## ERRORS

Access fails if one or more of the following are true:

- |           |  |
|-----------|--|
| [ENOTDIR] | A component of the <i>path</i> prefix is not a directory.  |
| [ENOENT]  | The named file does not exist.   |
| [EACCES]  | Search permission is denied on a component of the <i>path</i> prefix.  |
| [EROFS]   | Write access is requested for a file on a read-only file system.   |
| [ETXTBSY] | Write access is requested for a pure procedure (shared text) file that is being executed.  |
| [EACCES]  | Permission bits of the file mode do not permit the requested access.   |
| [EFAULT]  | <i>Path</i> points outside the allocated address space for the process. The reliable detection of this condition will be implementation dependent. |
| [EINVAL]  | The value of the <i>amode</i> argument is invalid.   |

## RETURN VALUE

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

chmod(2), stat(2), unistd(5), types(5).

### APPLICATIONS USAGE

The [EINVAL] error is taken from a SVID future direction. It may not be included in all implementations at present.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

SVID does not use symbolic names for *amode*. It does not, therefore, call for the inclusion of the header file <unistd.h>, and the description does not refer to this header file.

This change is forecast as a future direction in the SVID.



## NAME

acct — enable or disable process accounting (OPTIONAL)

## SYNOPSIS

```
int acct (path)
char *path;
```

## DESCRIPTION

*Acct* is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal, see *exit(2)* and *signal(2)*. The effective user ID of the calling process must be superuser to use this call.

*Path* points to a path name naming the accounting file. The format of an accounting file produced as a result of calling *acct(2)* has records in the format defined by the structure *acct* in *<sys/acct.h>*.

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

## ERRORS

*Acct* will fail if one or more of the following are true:

[EPERM]	The effective user of the calling process is not super-user.
[EBUSY]	An attempt is being made to enable accounting when it is already enabled.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	One or more components of the accounting file path name do not exist.
[EACCES]	The file named by <i>path</i> is not an ordinary file.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	<i>Path</i> points to an illegal address.

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*exit(2)*, *signal(2)*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

This *optional* function is included in the *kernel extension* set (K\_EXT) in the SVID.





## NAME

alarm — set a process alarm clock

## SYNOPSIS

unsigned alarm (sec)  
unsigned sec;

## DESCRIPTION

*Alarm* instructs the alarm clock of the calling process to send the signal SIGALRM to the calling process after the number of real time seconds specified by *sec* have elapsed, see *signal(2)*.

*Alarm* requests are not stacked; successive calls reset the alarm clock of the calling process.

If *sec* is 0, any previously made alarm request is cancelled.

*Fork(2)* sets the alarm clock of a new process to 0. A process created by *exec(2)* inherits the time left on the old process's alarm clock.

## RETURN VALUE

*Alarm* returns the amount of time previously remaining in the alarm clock of the calling process.

## SEE ALSO

*exec(2)*, *fork(2)*, *pause(2)*, *signal(2)*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

*brk*, *sbrk* — change data segment space allocation (OPTIONAL)

## SYNOPSIS

```
int brk (endds)
char *endds;

char *sbrk (incr)
int incr;
```

## DESCRIPTION

*Brk* and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment, see *exec(2)*. The change is made by resetting the process's break value and allocating the appropriate amount of space.

*Brk* sets the system's idea of the lowest data segment location not used by the program (called the "break") to *endds* (rounded to the convenient hardware addressing size). The amount of allocated space increases as the break value increases.

*Sbrk* adds *incr* bytes to the break value and changes the allocated space accordingly.

When a program begins execution via *exec(2)* the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use *sbrk*.

*Brk* sets the break value to *endds* and changes the allocated space accordingly.

*Sbrk* adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased. If *sbrk* is initially called with an *incr* of 0, then the value returned is the base of the existing data segment allocation.

When obtained, the data contents of the allocated region are undefined.

## ERRORS

*Brk* and *sbrk* will fail without making any change in the allocated space if the following is true:

[ENOMEM]	Such a change would result in more space being allocated than is allowed by a system-imposed maximum, see <i>ulimit(2)</i> .
----------	--

## RETURN VALUE

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, *brk* returns a value of -1 and *sbrk* returns a value of (*char \**) -1, and *errno* is set to indicate the error.

## SEE ALSO

*exec(2)*, *ulimit(2)*, *malloc(3X)*.

## APPLICATION USAGE

*Brk* may be called with any value in the range of memory addresses which have been returned by *sbrk*. The only real use for *brk* is to free a large amount of memory allocated by *sbrk*. If *brk* is used to allocate or free memory outside of this range, such usage may have undesirable effects. If an area is freed and subsequently reallocated, the contents of the area are not necessarily preserved.

*Malloc(3X)* is the recommended way to obtain additional working space. However, programs which use *malloc(3X)* or *stdio(3S)* should not make use of either *brk* or

*sbrk.*

**CHANGE HISTORY****Issue 1**

This *optional* function is not included in the SVID. It is taken from UNIX System V Release 2.0.



## NAME

chdir — change working directory

## SYNOPSIS

```
int chdir (path)
char *path;
```

## DESCRIPTION

*Path* points to the path name of a directory. *Chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with */*.

## ERRORS

*Chdir* will fail and the current working directory will be unchanged if one or more of the following are true:

- |           |   |
|-----------|---|
| [ENOTDIR] | A component of the path name is not a directory.  |
| [ENOENT]  | The named directory does not exist.   |
| [EACCES]  | Search permission is denied for any component of the path name.   |
| [EFAULT]  | <i>Path</i> points outside the allocated address space of the process. The reliable detection of this condition will be implementation dependent. |

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

chmod — change mode of file

## SYNOPSIS

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int chmod (path, mode)
```

```
char *path;
```

```
int mode;
```

## DESCRIPTION

*Path* points to a path name naming a file. *Chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are described in `<sys/stat.h>`, and are interpreted as follows:

S_ISUID†	04000	Set user ID on execution
S_ISGID	02000	Set group ID on execution
	01000	Reserved
S_IRUSR	00400	Read by owner
S_IWUSR	00200	Write by owner
S_IXUSR	00100	Execute (search if a directory) by owner
S_IRGRP	00040	Read by group
S_IWGRP	00020	Write by group
S_IXGRP	00010	Execute (search) by group
S_IROTH	00004	Read by others (i.e., anyone else)
S_IWOTH	00002	Write by others
S_IXOTH	00001	Execute (search) by others

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

For security reasons, if the effective user ID of the process is not super-user, and if the group ID of the file does not match the effective group ID, then a request to set the set-group-ID bit (S\_ISGID) is ignored.

## ERRORS

*Chmod* will fail and the file mode will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the <i>path</i> prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the <i>path</i> prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	<i>Path</i> points outside the allocated address space of the process. The reliable detection of this condition will be implementation dependent.

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of `-1` is returned and *errno* is set to indicate the error.

## SEE ALSO

chown(2), mknod(2), open(2), stat(5), types(5).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

†The use of symbolic names for *mode* has been introduced. This defines the values of access modes. As the SVID does not use symbolic names, it does not call for the inclusion of the header files `<sys/stat.h>` and `<sys/types.h>`, and the description does not refer to the `<sys/stat.h>` header file. (NB. This change is forecast in the SVID Future Directions Section.)

## Issue 2

The reference to the set-user-ID bit being cleared for security reasons if the user is not super-user has been removed and the remaining text of the paragraph on security issues has been clarified.

The **FUTURE DIRECTION** on mandatory or enforcement-mode file and record locking has been removed.



## NAME

`chown` — change owner and group of a file

## SYNOPSIS

```
int chown (path, owner, group)
char *path;
int owner, group;
```

## DESCRIPTION

*Path* points to a path name naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or superuser may change the ownership of a file.

For security reasons, if *chown* is successfully invoked by other than the superuser, the set-user-ID and set-group-ID bits of the file mode, `S_ISUID` and `S_ISGID` respectively, will be cleared.

## ERRORS

*Chown* will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

[ENOTDIR]	A component of the <i>path</i> prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the <i>path</i> prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	<i>Path</i> points outside the allocated address space of the process. The reliable detection of this condition will be implementation dependent.

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

`chmod(2)`, `stat(5)`.

## APPLICATION NOTE

Not all implementations allow users other than the super-user to change the owner or group of a file. Applications should not rely on the ability to do so.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

In the third paragraph of the **DESCRIPTION** section, the SVID does not use the symbolic values `S_ISUID` and `S_ISGID`, but instead refers to the specific bit values. (NB. This change is forecast in the SVID Future Directions section.)

### Issue 2

The word "successfully" was added to the final paragraph of the **DESCRIPTION** section to reflect changes made in Issue 2 of the SVID.

The **FUTURE DIRECTION** on mandatory or enforcement-mode file and record locking has been dropped.



## NAME

chroot — change root directory (OPTIONAL)

## SYNOPSIS

```
int chroot (path)
char *path;
```

## DESCRIPTION

*Path* points to a path name naming a directory. *Chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The user's working directory is unaffected by the *chroot* system call.

The effective user ID of the process must be super-user to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the subtree rooted at the root directory.

## ERRORS

*Chroot* will fail and the root directory will remain unchanged if one or more of the following are true:

- |           |  |
|-----------|--|
| [ENOTDIR] | Any component of the <i>path</i> name is not a directory.              |
| [ENOENT]  | The named directory does not exist.                                    |
| [EPERM]   | The effective user ID is not super-user.                               |
| [EFAULT]  | <i>Path</i> points outside the allocated address space of the process. |

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

chdir(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

This *optional* function is included in the *kernel extension* set (K\_EXT) in the SVID.





## NAME

close — close a file descriptor

## SYNOPSIS

```
int close (fildes)
int fildes;
```

## DESCRIPTION

*Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Close* closes the file descriptor indicated by *fildes*. All outstanding record locks on the file indicated by *fildes* that are owned by the calling process are removed.

## ERRORS

[EBADF] *Close* will return this error if *fildes* is not a valid open file descriptor.

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*creat*(2), *dup*(2), *exec*(2), *fcntl*(2), *open*(2), *pipe*(2).

## APPLICATIONS USAGE

Normally, applications should only use the *stdio* routines to open, close, read, and write files. Thus, an application that had used the *stdio* routine *fopen*(3S) to open a file would use the corresponding *fclose*(3S) routine rather than *close*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The SVID reads: "Close will fail if..." in the description of [EBADF].





## NAME

creat — create a new file or rewrite an existing one

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int creat (path, mode)
char *path;
int mode;
```

## DESCRIPTION

*Creat* creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID of the process; the group ID of the file is set to the effective group ID of the process; and the access permission bits (see *chmod(2)*) of the file mode are set to the value of *mode* modified as follows:

The corresponding bits are ANDed with the complement of the process's file mode creation mask, see *umask(2)*. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared.

Upon successful completion, the file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls, see *fcntl(2)*. No process may have more than {OPEN\_MAX} files open simultaneously. A new file may be created with a mode that forbids writing.

## ERRORS

*Creat* will fail if one or more of the following are true:

- |           |   |
|-----------|---|
| [EACCES]  | Search permission is denied on a component of the path prefix.  |
| [EACCES]  | The file does not exist and the directory in which the file is to be created does not permit writing. |
| [EACCES]  | The file exists and write permission is denied.   |
| [EFAULT]  | <i>Path</i> points outside the allocated address space of the process.                                |
| [EISDIR]  | The named file is an existing directory.  |
| [EMFILE]  | {OPEN_MAX} file descriptors are currently open in the calling process.                                |
| [ENFILE]  | The system file table is full. {SYS_OPEN} files are already open.                                     |
| [ENOENT]  | A component of the path name which must exist does not exist.   |
| [ENOSPC]  | The directory to contain the file cannot be extended.   |
| [ENOTDIR] | A component of the path prefix is not a directory.  |
| [EROFS]   | The named file resides or would reside on a read-only file system.                                    |
| [ETXTBSY] | The file is a pure procedure (shared text) file that is being executed.                               |

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

### SEE ALSO

`chmod(2)`, `close(2)`, `dup(2)`, `fcntl(2)`, `lseek(2)`, `open(2)`, `read(2)`, `umask(2)`, `write(2)`, `stat(5)`, `types(5)`.

### APPLICATIONS USAGE

Normally, applications should use the *stdio* routines to open, close, read and write files. In this case, *fopen(3S)* should be used rather than *creat(2)*.

*Creat* is now rendered obsolete by *open(2)* with `O_CREAT` set.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The SVID does not use symbolic names for the access permissions specified in the *mode* parameter. It therefore also does not call for the inclusion of `<sys/stat.h>` and `<sys/types.h>` and the description does not refer to the `<sys/stat.h>` header file. See *chmod(2)*.

#### Issue 2

Except that the **FUTURE DIRECTION** on mandatory or enforcement-mode file and record locking has been dropped, aligned with the entry in Issue 2 of the SVID by applying the following changes:

The words "complement of the" were added to the paragraph in the **DESCRIPTION** referring to the file mode creation mask.

The text "{SYS\_OPEN} files are already open." has been added in the description of the [ENFILE] error.



## NAME

dup — duplicate an open file descriptor

## SYNOPSIS

```
int dup (fildes)
int fildes;
```

## DESCRIPTION

*Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

The new file descriptor is set to remain open across *exec* system calls, see *fcntl*(2).

The file descriptor returned is the lowest one available.

## ERRORS

*Dup* will fail if one or more of the following are true:

[EBADF]            *Fildes* is not a valid open file descriptor.

[EMFILE]          {OPEN\_MAX} file descriptors are currently open in the calling process.

## RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*creat*(2), *close*(2), *exec*(2), *fcntl*(2), *open*(2), *pipe*(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

execl, execv, execlx, execve, execlp, execvp — execute a file

## SYNOPSIS

```
int execl (path, arg0, arg1, ..., argn, (char *)0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execlx (path, arg0, arg1, ..., argn, (char *)0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp)
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];
```

## DESCRIPTION

*Exec* in all its forms transforms the current process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header, a text segment, and a data segment. There can be no return from a successful *exec* because the calling process image is overlaid by the new process image.

When a C program is executed, it is called as follows:

```
main (argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to null-terminated strings that constitute the environment for the new process. *Argc* is conventionally at least one (1) and the initial member of the array points to a string containing the name of the file.

*Path* points to a path name that identifies the new process file. For *execlp* and *execvp*, *file* points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH=", see *environ*(5) and *system*(3S).

*Arg0*, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* is present and points to a string that is the same as *file* or *path* (or its last component).

*Argv* is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv[0]* points to a string that is the same as *file* or *path* (or its last component). *Argv* is terminated by a NULL pointer.

*Envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a NULL pointer. For *exec1* and *execv*, the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

```
extern char **environ;
```

and it is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl(2)*. For those file descriptors that remain open, the file pointer is unchanged.

Signals set to the default action (SIG\_DFL) in the calling process will be set to the default action in the new process. Signals set to be ignored (SIG\_IGN) by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to the default action in the new process, see *signal(2)*.

If the set-user-ID-on-execution mode bit of the new process file is set, see *chmod(2)*, *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID-on-execution mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process. The effective user ID and group ID of the new process are saved for use by *setuid(2)*.

If the set-user-ID (set-group-ID) mode bit of the new process file is not set, then the new process inherits the calling process's real and effective user (group) ID.

Profiling is disabled for the new process, see *profil(2)*. Profiling is an *optional* service.

Any shared memory segments attached to the calling process will not be attached to the new process, see *shmop(2)*. Shared memory is an *optional* service.

The new process also inherits at least the following attributes from the calling process:

- nice value (see *nice(2)*); *nice* is an *optional* service
- semadj* values (see *semop(2)*); semaphores are an *optional* service
- process ID
- parent process ID
- process group ID
- tty group ID (see *exit(2)* and *signal(2)*)
- trace flag (see *ptrace(2)* request 0); tracing is an *optional* service
- time left until an alarm clock signal (see *alarm(2)*)
- current working directory
- root directory
- file mode creation mask (see *umask(2)*)
- file size limit (see *ulimit(2)*)
- utime*, *stime*, *cutime*, and *cstime* (see *times(2)*)
- file locks, see *fcntl(2)* and *lockf(3C)*



## ERRORS

*Exec* will fail and return to the calling process if one or more of the following are true:

- |           |   |
|-----------|---|
| [ENOENT]  | One or more components of the path name of the new process file do not exist.   |
| [ENOTDIR] | A component of the new process file's path prefix is not a directory.   |
| [EACCES]  | Search permission is denied for a directory listed in the new process file's path prefix.   |
| [EACCES]  | The new process file is not an ordinary file, see <i>mknod</i> (2).   |
| [EACCES]  | The new process file mode denies execution permission.  |
| [ENOEXEC] | The <i>exec</i> is not an <i>exec/p</i> or <i>execvp</i> , and the new process file has the appropriate access permission but is not a valid executable object. |
| [ETXTBSY] | The new process file is a pure procedure (shared text) file that is currently open for writing by some process.   |
| [ENOMEM]  | The new process requires more memory than is allowed by the hardware or a system-imposed maximum.   |
| [E2BIG]   | The number of bytes in the new process argument list is greater than the system-imposed limit of {ARG_MAX} bytes.   |
| [EFAULT]  | The new process file image is corrupted.  |
| [EFAULT]  | <i>Path</i> points to an illegal address or <i>argv</i> or <i>envp</i> point to an illegal address, directly or indirectly.                                     |

## RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value will be -1 and *errno* will be set to indicate the error.

## SEE ALSO

*alarm*(2), *exit*(2), *fcntl*(2), *fork*(2), *mknod*(2), *nice*(2), *profil*(2), *ptrace*(2), *semop*(2), *shmop*(2), *signal*(2), *times*(2), *ulimit*(2), *umask*(2), *lockf*(3C).

## APPLICATIONS USAGE

If possible, applications should use *system*(3S), which is easier to use and supplies more functions, rather than *fork*(2) and *exec*(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

### Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following changes:

The **DESCRIPTION** has been clarified and includes mention of file locks.

The effects of the interprocess communication options have been added.

For convenience of the user, the relationships between *exec(2)* and the optional facilities — interprocess communication, *nice(2)*, *profil(2)*, *ptrace(2)* — have been given on these sheets. In Issue 2 of the SVID these options are all included in the *kernel extension* set, and their effect on *exec* is given in a separate section.



## NAME

`exit`, `_exit` — terminate process

## SYNOPSIS

`void exit (status)`

`int status;`

`void _exit (status)`

`int status;`

## DESCRIPTION

*Exit* terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait(2)*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it; see *wait(2)*. If the parent is not waiting, the child's status will be made available to it when the parent subsequently executes *wait(2)*.

If the parent process of the calling process is not executing a *wait(2)*, the calling process is transformed into a *zombie process*. A *zombie process* is an inactive process and it will be deleted at some later time when its parent process executes *wait(2)*.

The parent process ID of all of the calling process' existing child processes and zombie processes is set to the process ID of a special system process. That is, these processes are inherited by a special system process.

If the process has a process, text or data lock, an *unlock* is performed; see *plock(2)*. Process locking is an *optional* service.

Each attached shared memory segment is detached and the value of *shm\_nattch* (see *shmget(2)*) in the data structure associated with its shared memory ID is decremented by 1. Shared memory is an *optional* service.

For each semaphore for which the calling process has set a *semadj* value, see *semop(2)*, that value is added to the *semval* of the specified semaphore. Semaphores are an *optional* service.

An accounting record is written on the accounting file if the system's accounting routine is enabled; see *acct(2)*. Accounting is an *optional* service.

If the process is a process group leader, and is a member of a terminal group, the *SIGHUP* signal is sent to each process that has a process group ID and tty group ID equal to that of the calling process.

The function *exit* may cause cleanup actions, see *fclose(3S)* before the process exits. The function *\_exit* circumvents all cleanup.

## RETURN VALUE

These routines do not return a value.

## SEE ALSO

*acct(2)*, *plock(2)*, *semop(2)*, *shmget(2)*, *signal(2)*, *wait(2)*, *fclose(3S)*.

### APPLICATION USAGE

Normally applications should use *exit* rather than *\_exit*.

Not only do these routines not return a value, they do not return at all.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID.

#### Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following changes:

The description of the conditions leading to sending of **SIGHUP** has been clarified.

A description of the effects of the interprocess communication options has been added.

For the convenience of the user, the interactions between *exit* and the *optional* facilities — interprocess communication, *plock(2)* and *acct(2)* — have been identified in the **DESCRIPTION** section. In the SVID these *optional* facilities are included in the *kernel extension* set, and their effect on *exit* is given in a separate section.



## NAME

fcntl — file control

## SYNOPSIS

```
#include <unistd.h>
#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd;
```

## DESCRIPTION

*Fcntl* provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Arg* value and type are specific to the type of command.

The *cmd* values available are:

- F\_DUPFD** Return a new file descriptor as follows:
- Lowest numbered available file descriptor greater than or equal to *arg*.
  - Same open file (or pipe) as the original file.
  - Same file pointer as the original file (i.e., both file descriptors share one file pointer).
  - Same access mode (read, write or read/write).
  - Same file status flags (i.e., both file descriptors share the same file status flags).
  - The close-on-exec flag associated with the new file descriptor is set to remain open across *exec(2)* system calls.
- F\_GETFD** Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is 0 the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.
- F\_SETFD** Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (0 or 1 as above).
- F\_GETFL** Get file status flags: *O\_RDONLY*, *O\_WRONLY*, *O\_RDWR*, *O\_NDELAY*, *O\_APPEND*.
- F\_SETFL** Set file status flags to *arg*. Only certain flags can be set, see *fcntl(5)*.
- F\_GETLK** Get the first lock which blocks the lock description given by the variable of type *struct flock* pointed to by *arg* (see below). The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to *F\_UNLCK*.
- F\_SETLK** Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* (see below). The *cmd* *F\_SETLK* is used to establish read (*F\_RDLCK*) and write (*F\_WRLCK*) locks, as well as remove either type of lock (*F\_UNLCK*). If a read or write lock cannot be set, *fcntl* will return immediately with an error value of -1.
- F\_SETLKW** This *cmd* is the same as *F\_SETLK* except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free

to be locked.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure *flock* describes the type (*l\_type*), starting offset (*l\_whence*), relative offset (*l\_start*), size (*l\_len*) and process ID (*l\_pid*) of the segment of the file to be affected.

*l\_whence* is  $\dagger$ SEEK\_SET, SEEK\_CUR, SEEK\_END, to indicate that the relative offset will be measured from the start of the file, current position or end of the file, respectively.

The process ID and system ID fields are only used with the *F\_GETLK cmd* to return the value for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l\_len* to zero (0). If such a lock also has *l\_start* set to zero (0), the whole file will be locked. Changing or unlocking a portion from the middle of a larger locked segment leaves a smaller segment at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork(2)* system call.

## ERRORS

*Fcntl* will fail if one or more of the following are true:

[EBADF]	<i>Fildes</i> is not a valid open file descriptor.
[EMFILE]	<i>Cmd</i> is <i>F_DUPFD</i> and { <i>OPEN_MAX</i> } file descriptors are currently open in the calling process.
[EINVAL]	<i>Cmd</i> is <i>F_DUPFD</i> and <i>arg</i> is negative or greater than or equal to { <i>OPEN_MAX</i> }.
[EINVAL]	<i>Cmd</i> is <i>F_GETLK</i> , <i>F_SETLK</i> , or <i>F_SETLKW</i> and <i>arg</i> or the data it points to is not valid.
[EACCES] [EAGAIN]	Due to implementation details, one of these errors will be returned for the following condition: <i>cmd</i> is <i>F_SETLK</i> ; the type of lock ( <i>l_type</i> ) is a read ( <i>F_RDLCK</i> ) or write ( <i>F_WRLCK</i> ) lock and the segment of a file to be locked is already write locked by another process, or the type is a write lock and the segment of a file to be locked is already read or write locked by another process.
[ENOLCK]	<i>Cmd</i> is <i>F_SETLK</i> or <i>F_SETLKW</i> , the type of lock is a read or write lock and there are no more file locks available (too many segments are locked).



[EDEADLK] *Cmd* is F\_SETLKW, the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free, would cause a deadlock.

[EDEADLK] *Cmd* is F\_SETLKW and changing or unlocking a portion of a larger locked segment would require additional locks, exceeding the number in the system lock table, {LCK\_MAX}.

### RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor
F_GETFD	Value of flag (only the low-order bit is defined)
F_SETFD	Value other than -1
F_GETFL	Value of file flags
F_SETFL	Value other than -1
F_GETLK	Value other than -1
F_SETLK	Value other than -1
F_SETLKW	Value other than -1

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

### SEE ALSO

close(2), exec(2), open(2), lockf(3C), fcntl(5).

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

†The SVID uses the absolute values 0, 1 and 2 for *l\_whence* instead of the symbolic values SEEK\_SET, SEEK\_CUR and SEEK\_END. These names come from the <unistd.h> file described in Appendix BASE: 1.6, Comparison to the 1984 /usr/group Standard.

#### Issue 2

Except that the **FUTURE DIRECTION** on mandatory or enforcement-mode file and record locking has been dropped, aligned with Issue 2 of the SVID by applying the following changes:

The reference to [EACCES] has been included.

The command causing the [EDEADLK] error has been corrected from F\_SETLK to F\_SETLKW, and the condition that changing or unlocking a portion of a larger segment would require additional locks has been added to this error.

The description of *l\_sysid* has been removed. It was included in Issue 1 in error.





## NAME

fork — create a new process

## SYNOPSIS

int fork ( )

## DESCRIPTION

*Fork* causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

- environment
- close-on-exec flag (see *exec(2)*)
- signal handling settings (i.e., *SIG\_DFL*, *SIG\_IGN*, function address)
- set-user-ID mode bit
- set-group-ID mode bit
- profiling on/off status (see *profil(2)*; profiling is an *optional* service)
- nice value (see *nice(2)*; *nice* is an *optional* service)
- process group ID
- all attached shared memory segments (see *shmop(2)*; shared memory is an *optional* service)
- tty group ID (see *exit(2)* and *signal(2)*)
- trace flag (see *ptrace(2)* request 0; *ptrace* is an *optional* service)
- current working directory
- root directory
- file mode creation mask (see *umask(2)*)
- file size limit (see *ulimit(2)*)

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

Process locks, text locks and data locks are not inherited by the child (see *plock(2)*; process locking is an *optional* service).

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

All *semadj* values are cleared (see *semop(2)*; semaphores are an *optional* service).

File locks set by the parent process are not inherited by the child process, see *fcntl(2)* or *lockf(3C)*.

### ERRORS

*Fork* will fail and no child process will be created if one or more of the following are true:

- [EAGAIN] The system-imposed limit on the total number of processes under execution system-wide {PROC\_MAX} or by a single user ID {CHILD\_MAX} would be exceeded.
- [ENOMEM] The process requires more space than the system is able to supply.

### RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

### SEE ALSO

*exec*(2), *exit*(2), *fcntl*(2), *nice*(2), *plock*(2), *profil*(2), *ptrace*(2), *semop*(2), *shmop*(2), *signal*(2), *times*(2), *ulimit*(2), *umask*(2), *wait*(2), *lockf*(3C).

### APPLICATION USAGE

If possible, applications should use *system*(3S), which is easier to use and supplies more functions, rather than *fork*(2) and *exec*(2).

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID.

#### Issue 2

Aligned with Issue 2 of the SVID by applying the following changes:

The paragraph on file locking has been added.

The effects of the interprocess communication options have been added.

For user convenience, the interactions between *fork* and the *optional* facilities — interprocess communication, *profil*(2), *ptrace*(2), *plock*(2) and *nice*(2) — have been included in the **DESCRIPTION** section. In Issue 2 of the SVID these *optional* facilities are included in the *kernel extension* set, and their effect on *fork* is given in a separate section.



NAME

getpid, getpgrp, getppid — get process, process group, and parent process IDs

SYNOPSIS

int getpid ( )

int getpgrp ( )

int getppid ( )

DESCRIPTION

*Getpid* returns the process ID of the calling process.

*Getpgrp* returns the process group ID of the calling process.

*Getppid* returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), setpgrp(2), signal(2).

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID.





NAME

getuid, geteuid, getgid, getegid — get real user, effective user, real group, and effective group IDs

SYNOPSIS

unsigned short getuid ( )

unsigned short geteuid ( )

unsigned short getgid ( )

unsigned short getegid ( )

DESCRIPTION

*Getuid* returns the real user ID of the calling process.

*Geteuid* returns the effective user ID of the calling process.

*Getgid* returns the real group ID of the calling process.

*Getegid* returns the effective group ID of the calling process.

SEE ALSO

setuid(2).

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

`ioctl` — control device

## SYNOPSIS

```
int ioctl (fildes, request, arg)
int fildes, request;
```

## DESCRIPTION

`ioctl` performs a variety of functions on devices, typically character special files. *Fildes* is an open file descriptor. *Request* selects the function to be performed and will depend on the device being addressed. *Arg* value and type are specific to the device and request.

## ERRORS

`ioctl` will fail if one or more of the following are true:

- |          |   |
|----------|---|
| [EBADF]  | <i>Fildes</i> is not a valid open file descriptor.                            |
| [ENOTTY] | <i>Fildes</i> is not associated with a device that accepts control functions. |
| [EINVAL] | <i>Request</i> or <i>arg</i> is not valid.                                    |
| [EINTR]  | A signal was caught during the <code>ioctl</code> operation.                  |

## RETURN VALUE

If an error has occurred, a value of -1 is returned and *errno* is set to indicate the error.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following changes:

In the SVID, the last sentence of the DESCRIPTION section reads: "*Arg* also is specific to the device and request", and the error [ENOTTY] reads: "*fildes* is not associated with a character special device".





## NAME

kill — send a signal to a process or a group of processes

## SYNOPSIS

```
#include <signal.h>
int kill (pid, sig)
int pid;
int sig;
```

## DESCRIPTION

*Kill* sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal(2)*, or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is super-user.

The processes with a process ID less than or equal to {SYSPID\_MAX} are *special processes*.

If *pid* is greater than 0, *sig* will be sent to the process whose process ID is equal to *pid*. †If a signal is sent to a *special process*, the effect is implementation defined.

If *pid* is 0, *sig* will be sent to all processes, excluding the *special processes*, whose process group ID is equal to the process group ID of the sender.

If *pid* is -1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes, excluding the *special processes*, whose real user ID is equal to the effective user ID of the sender.

If *pid* is -1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding the *special processes*.

If *pid* is negative but not -1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

## ERRORS

*Kill* will fail and no signal will be sent if one or more of the following are true:

- |          |  |
|----------|--|
| [EINVAL] | <i>Sig</i> is not a valid signal number.   |
| [EPERM]  | <i>Sig</i> is SIGKILL and <i>pid</i> is less than or equal to {SYSPID_MAX}.  |
| [ESRCH]  | No process can be found corresponding to that specified by <i>pid</i> .  |
| [EPERM]  | The user ID of the sending process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process. |

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

getpid(2), setpgid(2), signal(2), signal(5).

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID with the following changes:

The sentence "The processes with a process ID less than {SYSPID\_MAX} are *special processes*" is additional to the SVID wording. The SVID treats {SYSPID\_MAX} as 1.

†The second sentence of the paragraph starting "If *pid* is greater than 0..." is additional to the SVID.

The introduction of the error [EPERM] is forecast in the Future Directions section in the SVID.



## NAME

link — link to a file

## SYNOPSIS

```
int link (path1, path2)
char *path1, *path2;
```

## DESCRIPTION

*Path1* points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

## ERRORS

*Link* will fail and no link will be created if one or more of the following are true:

- |           |  |
|-----------|--|
| [ENOTDIR] | A component of either path prefix is not a directory.  |
| [ENOENT]  | A component of either path name which must exist does not exist.   |
| [EACCES]  | A component of either path prefix denies search permission.  |
| [EEXIST]  | The link named by <i>path2</i> exists.   |
| [EPERM]   | The file named by <i>path1</i> is a directory and the effective user ID is not super-user.   |
| [EXDEV]   | The link named by <i>path2</i> and the file named by <i>path1</i> are on different logical devices (file systems) and the implementation does not permit cross device links. |
| [EACCES]  | The requested link requires writing in a directory with a mode that denies write permission.   |
| [EROFS]   | The requested link requires writing in a directory on a read-only file system.   |
| [EMLINK]  | The maximum number of links to a file would be exceeded.   |
| [EFAULT]  | <i>Path</i> points outside the allocated address space of the process.   |
| [ENOSPC]  | The directory to contain the file cannot be extended.  |

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

unlink(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

`lseek` — move read/write file pointer

## SYNOPSIS

```
#include <unistd.h>

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;
```

## DESCRIPTION

*Fildes* is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *Lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is `SEEK_SET` (0), the pointer is set to *offset* bytes.

If *whence* is `SEEK_CUR` (1), the pointer is set to its current location plus *offset*.

If *whence* is `SEEK_END` (2), the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned.

## ERRORS

*Lseek* will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *Fildes* is not an open file descriptor.

[ESPIPE] *Fildes* is associated with a pipe or fifo.

[EINVAL] and SIGSYS signal.  
*Whence* is not one of the valid numbers.

[EINVAL] The resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

## RETURN VALUE

Upon successful completion, a file pointer value is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

`creat(2)`, `dup(2)`, `fcntl(2)`, `open(2)`, `types(5)`, `unistd(5)`.

## APPLICATION USAGE

Normally, applications should use the *stdio* library routines to open, close, read, write and manipulate files. Thus, an application that had used the *stdio* routine *fopen*(3S) to open a file would use *fseek*(3S) rather than *lseek*(2).

Reliance should not be placed on receipt of the SIGSYS signal in the case that the value of *whence* is not valid. SIGSYS has been removed from SVID Issue 2 and may be subject to later withdrawal.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID with the following changes:

The SVID uses absolute values rather than the symbolic values `SEEK_SET`, `SEEK_CUR`, `SEEK_END` for *whence*. It therefore also does not call for the inclusion of `<unistd.h>`. This change is forecast in the SVID Future Directions section, in the *lseek*(OS) entry. It also causes a minor wording change for the error "[EINVAL] and SIGSYS signal".

The error "[EINVAL]: The resulting file pointer would be negative" is additional wording to the SVID.

#### Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

The **APPLICATION USAGE** relating to **SIGSYS** has been added.



## NAME

mknod — make a directory, or a special or ordinary file

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mknod (path, mode, dev)
char *path;
int mode, dev;
```

## DESCRIPTION

*Mknod* creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*, where the value of *mode* is interpreted as follows:

File type; one of the following:

S_IFIFO†	0010000	fifo special
S_IFCHR	0020000	character special
S_IFDIR	0040000	directory
S_IFBLK	0060000	block special
S_IFREG	0100000	ordinary file
	0000000	ordinary file

Special execution bits; constructed from the following:

S_ISUID	0004000	set user ID on execution
S_ISGID	0002000	set group ID on execution
	0001000	reserved

Access permissions; constructed from the following:

S_IRWXU	0000700	read, write, execute (search) by owner
S_IRUSR	0000400	read by owner
S_IWUSR	0000200	write by owner
S_IXUSR	0000100	execute (search on directory) by owner
S_IRWXG	0000070	read, write, execute (search) by group
S_IRGRP	0000040	read by group
S_IWGRP	0000020	write by group
S_IXGRP	0000010	execute (search on directory) by group
S_IRWXO	0000007	read, write, execute (search) by others
S_IROTH	0000004	read by others
S_IWOTH	0000002	write by others
S_IXOTH	0000001	execute (search on directory) by others

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process.

Values of *mode* other than those above are undefined and should not be used. The owner, group and other permission bits of *mode* are modified by the process's file mode creation mask: all bits whose corresponding bit in the process's file mode creation mask is set are cleared, see *umask(2)*. If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special file, *dev* is ignored.

*Mknod* may be invoked only with the effective user ID of the super-user for file types other than fifo special.

## ERRORS

*Mknod* will fail and the new file will not be created if one or more of the following are true:

[EPERM]	The effective user ID of the process is not super-user and the file type is not fifo special.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	A component of the path prefix denies search permission.
[EROFS]	The directory in which the file is to be created is located on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	<i>Path</i> points outside the process's allocated address space.
[ENOSPC]	The directory which would contain the new file cannot be extended.

## RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

chmod(2), exec(2), pipe(2), stat(2), umask(2), stat(5), types(5).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following changes:

† The SVID does not use the symbolic names for *mode*; it only gives the absolute values. It therefore also does not call for the inclusion of `<sys/stat.h>` and `<sys/types.h>`. The "Appendix BASE: 1.6 Comparison to the 1984 /usr/group Standard" part of the SVID shows these names as a future direction for the `<stat.h>` header file.

In the SVID, the mode value 0001000 is identified as "save text image after execution" instead of being "reserved".

## Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

Issue 2 of the SVID now reserves the mode value 0001000. The SVID does still not use symbolic names for *mode*.



## NAME

mount — mount a file system

## SYNOPSIS

```
int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;
```

## DESCRIPTION

*Mount* requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if 1, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

*Mount* may be invoked only with an effective user ID of the super-user.

## ERRORS

*Mount* will fail if one or more of the following are true:

- |           |   |
|-----------|---|
| [EPERM]   | The effective user ID is not super-user.  |
| [ENOENT]  | Any of the named files does not exist.  |
| [ENOTDIR] | A component of a path prefix is not a directory.  |
| [ENOTBLK] | <i>Spec</i> is not a block special device.  |
| [ENXIO]   | The device associated with <i>spec</i> does not exist.  |
| [ENOTDIR] | <i>Dir</i> is not a directory.  |
| [EFAULT]  | <i>Spec</i> or <i>dir</i> points outside the allocated address space of the process.              |
| [EBUSY]   | <i>Dir</i> is currently mounted on, is someone's current working directory, or is otherwise busy. |
| [EBUSY]   | The device associated with <i>spec</i> is currently mounted.                                      |
| [EBUSY]   | There are no more mount table entries.  |

## RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

umount(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The last sentence of the **DESCRIPTION**, in the SVID reads: " *Mount* may be invoked only by the super-user."





## NAME

*nice* — change priority of a process (OPTIONAL)

## SYNOPSIS

```
int nice (incr)
int incr;
```

## DESCRIPTION

*Nice* adds the value of *incr* to the *nice* value of the calling process. A process's *nice value* is a non-negative number for which a more positive value results in lower CPU priority.

A maximum *nice* value of 39 and a minimum *nice* value of 0 are imposed by the system. Requests for values above or below these limits result in the *nice* value being set to the corresponding limit.

## ERRORS

[EPERM] *Nice* will fail and not change the *nice* value if *incr* is negative or greater than 40 and the effective user ID of the calling process is not super-user.

## RETURN VALUE

Upon successful completion, *nice* returns the new *nice* value minus 20. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*exec*(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

This *optional* facility is included in the SVID *kernel extension* set (K\_EXT).





## NAME

open — open for reading or writing

## SYNOPSIS

```
#include <fcntl.h>
```

```
'int open (path, oflag [ , mode ] )
char *path;
int oflag, mode;
```

## DESCRIPTION

*Path* points to a path name naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by OR-ing flags from the following list (of the first three flags only one may be used):

O\_RDONLY    Open for reading only.

O\_WRONLY    Open for writing only.

O\_RDWR      Open for reading and writing.

O\_NDELAY     This flag may affect subsequent reads and writes, see *read(2)* and *write(2)*.

When opening a fifo with O\_RDONLY or O\_WRONLY set:

If O\_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O\_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line: (see CAVEATS, Chapter 1, discussion of *termio* ).

If O\_NDELAY is set:

The open will return without waiting for carrier.

If O\_NDELAY is clear:

The open will block until carrier is present.

O\_APPEND    If set, the file pointer will be set to the end of the file prior to each write.

O\_CREAT     If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the effective group ID of the process, and the access permission bits (see *chmod(2)*) of the file mode are set to the value of *mode* modified as follows, see *creat(2)*:

The corresponding bits are ANDed with the complement of the process's file mode creation mask. See *umask(2)*. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared.

**O\_TRUNC** If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

**O\_EXCL** If **O\_EXCL** and **O\_CREAT** are set, *open* will fail if the file exists.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is the lowest-numbered file-descriptor available and is set to remain open across *exec* system calls, see *fcntl(2)*.

## ERRORS

The named file is opened unless one or more of the following are true:

- [EACCES] A component of the path prefix denies search permission or the file does not exist and the directory in which the file is to be created does not permit writing.
- [EACCES] *Oflag* permission is denied for the named file.
- [EEXIST] **O\_CREAT** and **O\_EXCL** are set, and the named file exists.
- [EFAULT] *Path* points outside the allocated address space of the process.
- [EINTR] A signal was caught during the *open* system call.
- [EISDIR] The named file is a directory and *oflag* is write or read/write.
- [EMFILE] {**OPEN\_MAX**} file descriptors are currently open.
- [ENFILE] The system file table is full. {**SYS\_OPEN**} files are open.
- [ENOENT] **O\_CREAT** is not set and the named file does not exist. A component of the path name which must exist does not exist.
- [ENOSPC] The directory which would contain the new file cannot be expanded, the file does not exist, and **O\_CREAT** is specified.
- [ENOTDIR] A component of the path prefix is not a directory.
- [ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.
- [ENXIO] **O\_NDELAY** is set, the named file is a fifo, **O\_WRONLY** is set and no process has the file open for reading.
- [EROFS] The named file resides on a read-only file system and *oflag* is write or read/write.
- [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.

Upon successful completion, the file descriptor is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*chmod(2)*, *close(2)*, *creat(2)*, *dup(2)*, *fcntl(2)*, *lseek(2)*, *read(2)*, *write(2)*, *umask(2)*, *fcntl(5)*.



## APPLICATION USAGE

Normally applications should use the *stdio* routines to open, close, read and write files. Thus applications should use the *stdio* routine *fopen*(3S) rather than using *open*(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

The first paragraph of the **DESCRIPTION** has been clarified.

Except that the **FUTURE DIRECTION** on mandatory or forced-mode file locking has been dropped, aligned with Issue 2 of the SVID by applying the following changes:

The words "complement of the" were added to the paragraph of the **DESCRIPTION** referring to the file mode creation mask. They were omitted in error from Issue 1.

The words "the lowest-numbered file-descriptor available and is" have been included to reflect a change made in Issue 2 of the SVID.





## NAME

pause — suspend process until signal is received

## SYNOPSIS

int pause ( )

## DESCRIPTION

*Pause* suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

## ERRORS

[EINTR] See RETURN VALUE below.

## RETURN VALUE

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function see *signal(2)*, the calling process resumes execution from the point of suspension; *pause* returns a value of -1 and *errno* is set to [EINTR].

## SEE ALSO

alarm(2), kill(2), signal(2), wait(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

pipe — create an interprocess channel

## SYNOPSIS

```
int pipe (fildes)
int fildes[2];
```

## DESCRIPTION

*Pipe* creates an I/O mechanism called a *pipe* and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing and the O\_NDELAY flag is cleared.

Up to {PIPE\_MAX} bytes of data are buffered by the pipe before the writing process is blocked. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out basis.

## ERRORS

*Pipe* will fail if one or more of the following are true:

[EMFILE]            If {OPEN\_MAX} - 1 or more file descriptors are currently open in this process.

[ENFILE]            The system file table would overflow.

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

read(2), write(2).

## FUTURE DIRECTIONS

[EFAULT] will be returned in *errno* if the argument is not a valid address for this process.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

plock — lock process, text, or data in memory (OPTIONAL)

## SYNOPSIS

```
#include <sys/lock.h>
```

```
int plock (op)
```

```
int op;
```

## DESCRIPTION

*Plock* allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *Plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *Op* specifies the following:

PROCLCK	lock text and data segments into memory (process lock)
TXTLCK	lock text segment into memory (text lock)
DATLCK	lock data segment into memory (data lock)
UNLOCK	remove locks

## ERRORS

*Plock* will fail and not perform the requested operation if one or more of the following are true:

[EPERM]	The effective user ID of the calling process is not super-user.
[EINVAL]	<i>Op</i> is equal to PROCLCK and a process lock, a text lock, or a data lock already exists on the calling process.
[EINVAL]	<i>Op</i> is equal to TXTLCK and a text lock, or a process lock already exists on the calling process.
[EINVAL]	<i>Op</i> is equal to DATLCK and a data lock, or a process lock already exists on the calling process.
[EINVAL]	<i>Op</i> is equal to UNLOCK and no type of lock exists on the calling process.

## RETURN VALUE

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

exec(2), exit(2), fork(2), lock(5).

## APPLICATION USAGE

*Plock(2)* should not be used by most applications. Only programs that must have the type of real-time control it provides should use it.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

This *optional* facility is included in the SVID *kernel extension* set (K\_EXT).



## NAME

profil — execution time profile (OPTIONAL)

## SYNOPSIS

```
void profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;
```

## DESCRIPTION

*Buff* points to an area of store whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick ({CLK\_TCK} times per second); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to an entry inside *buff*, that entry is incremented. An "entry" is defined as a series of bytes with length *sizeof(short)*.

The interpretation of *scale* is implementation defined.

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec(2)* is executed, but remains on in child and parent both after a *fork(2)*. Profiling will be turned off if an update in *buff* would cause a memory fault.

## RETURN VALUE

Not defined.

## SEE ALSO

*exec(2)*, *fork(2)*.

## APPLICATION USAGE

*Profil* would normally be used in an application program only during its development, to analyse the program's performance.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID except that the SVID gives an explicit interpretation of *scale* as follows:

"The *scale* is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(octal) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock)."

This *optional* facility is included in the SVID *kernel extension* set (K\_EXT).





## NAME

ptrace — process trace (OPTIONAL)

## SYNOPSIS

```
int ptrace (request, pid, addr, data);
int request, pid, data;
```

## DESCRIPTION

*Ptrace* provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging. The child process behaves normally until it encounters a signal (see *signal(2)*) at which time it enters a stopped state and its parent is notified via *wait(2)*. When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The data type of the argument *addr* depends on the particular *request* given to *ptrace*.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

- 0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func*, see *signal(2)*. The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

- 1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If instruction (I) and data (D) space are separated, request 1 returns a word from I space, and request 2 returns a word from D space. If I and D space are not separated either request 1 or request 2 may be used with equal results. The *data* argument is ignored. These two requests will fail if *addr* is not the start address of a word, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to [EIO].
- 3 With this request, the word at location *addr* in the child's USER area in the system's address space is returned to the parent process. The *data* argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to [EIO].
- 4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated, request 4 writes a word into I space, and request 5 writes a word into D space. If I and D space are not separated, either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of



the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. Upon failure a value of -1 is returned to the parent process and the parent's *errno* is set to [EIO].

- 6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. Entries that can be written are implementation specific but might include general registers of the Processor Status Word.
- 7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent. This request will fail if *data* is not 0 or a valid signal number, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to [EIO].
- 8 This request causes the child to terminate with the same consequences as *exit(2)*.
- 9 This request is implementation dependent but if operative, it is used to request single-stepping through the instructions of the child.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec(2)* calls. If a traced process calls *exec*, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

## ERRORS

*Ptrace* will in general fail if one or more of the following are true:

- |         |   |
|---------|---|
| [EIO]   | <i>Request</i> is an illegal number. See the summary for each request type above.                     |
| [ESRCH] | <i>Pid</i> identifies a child that does not exist or has not executed a <i>ptrace</i> with request 0. |

## RETURN VALUE

Upon failure, a value of -1 is returned, and *errno* is set as above. Return values on successful completion are specific to the request type (see above).

## SEE ALSO

*exec(2)*, *exit(2)*, *signal(2)*, *wait(2)*.

## APPLICATION USAGE

*Ptrace(2)* should not be used by applications. It is only used by software debugging programs and it is hardware dependent.

Parts of this may not be implementable on some hardware; other hardware may require that it be extended.

The terms "word", "location", "start address", etc., used in the DESCRIPTION, are only meaningful within the context of the particular hardware.



CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID.

This *optional* facility is included in the SVID *kernel extension* set (K\_EXT).

Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

The data type of *addr* has been altered from *int* to variable, and a paragraph describing *addr* has been added.



## NAME

read — read from file

## SYNOPSIS

```
int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

## DESCRIPTION

*Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

*Read* attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking, e.g., terminals, always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl(2)* and *termio(7)*) or if the number of bytes left in the file is less than *nbyte* bytes or if the file is a pipe or a special file. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an empty pipe (or fifo):

If *O\_NDELAY* is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a character special file that has no data currently available:

If *O\_NDELAY* is clear, the read will block until the data becomes available. (This functionality of *O\_NDELAY* depends on the implementation of the *termio(7)* interface† see CAVEATS, Chapter 1).

The function *read* reads data previously written to a file. If any portion of an ordinary file prior to the end-of-file has not been written, the function *read* returns bytes with value 0. For example, the *lseek(2)* routine allows the file pointer to be set beyond the end of existing data in the file. If data are later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data are written into the gap.

## ERRORS

*Read* will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid file descriptor open for reading.

[EFAULT] *Buf* points outside the allocated address space.

[EINTR] A signal was caught during the *read* system call.

[EIO] An I/O error occurred on a special file.

[ENXIO] A request was made of a non-existent special file, or the request was outside the capabilities of the device.



### RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a -1 is returned and *errno* is set to indicate the error.

### SEE ALSO

*creat*(2), *dup*(2), *fcntl*(2), *ioctl*(2), *lseek*(2), *open*(2), *pipe*(2), *signal*(2), *lockf*(3C), *termio*(7).

### APPLICATION USAGE

Normally, applications should use the *stdio* library routines to open, close, read, and write files. Thus, an application that used the *stdio* routine *fopen*(3S) to open a file should use the *stdio* routine *fread*(3S) rather than *read*(2) to read it.

### FUTURE DIRECTIONS

*Read* on a pipe, fifo, or *tty* line with the *O\_NDELAY* flag set will return -1 rather than 0 when no data was present at the time of the *read*.

[EAGAIN] will be returned in *errno* when no data is available on a pipe, fifo or *tty* line being read.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID.

† This sentence additional to SVID.

‡ Subject to restrictions imposed by functionality of *termio*(7).

#### Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following changes:

The SVID has incorporated the errors [EIO] and [ENXIO] .

The description of reading unwritten portions of a file has been taken from Issue 2 of the SVID.

The FUTURE DIRECTION on mandatory or enforcement mode file and record locking has been dropped.

NAME

setpgrp — set process group ID

SYNOPSIS

int setpgrp ( )

DESCRIPTION

*Setpgrp* sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

Unless the process is already a process group leader, *setpgrp* disassociates the process from the terminal group, if any.

RETURN VALUE

*Setpgrp* returns the value of the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), kill(2), signal(2).

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID.

Issue 2

The paragraph on disassociation from a terminal group has been added as this describes the behaviour of System V Release 2.0.





## NAME

setuid, setgid — set user and group IDs

## SYNOPSIS

```
int setuid (uid)
```

```
int uid;
```

```
int setgid (gid)
```

```
int gid;
```

## DESCRIPTION

*Setuid (setgid)* is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid (gid)*.

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid (gid)*, the effective user (group) ID is set to *uid (gid)*.

If the effective user ID of the calling process is not super-user, but the saved set-user ID from *exec(2)* is equal to *uid*, the effective user ID is set to *uid*.

## ERRORS

[EPERM] *Setuid (setgid)* will fail if the real user (group) ID of the calling process is not equal to *uid (gid)* and its effective user ID is not super-user.

[EINVAL] The *uid (gid)* is out of range.

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*exec(2)*, *getuid(2)*.

## APPLICATION USAGE

The type of the argument taken by *setuid* differs from the type returned by *getuid(2)*. This may prompt diagnostics from *lint* (see "C LANGUAGE") but is otherwise harmless.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following changes:

The third paragraph of the **DESCRIPTION**, which refers to action taken if the effective user ID of the calling process is not super-user, has been added as it was erroneously omitted from Issue 1.

References to the saved set-group-ID have been deleted.



## NAME

signal — specify what to do upon receipt of a signal

## SYNOPSIS

```
#include <signal.h>

int (*signal (sig, func)) ( )
int sig;
int (*func) ( );
```

## DESCRIPTION

*Signal* allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

*Sig* can be assigned any one of the following except SIGKILL:

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT†	quit
SIGILL†	illegal instruction (not reset when caught)
SIGTRAP†	trace trap (not reset when caught)
SIGABRT	process abort signal
SIGFPE†	floating point exception
SIGKILL	kill (cannot be caught or ignored)
SIGSEGV†	segmentation violation
SIGSYS†	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2

† The default action for these signals is an abnormal process termination. See SIG\_DFL below.

For portability, applications should use the signals listed above and no others. (For example, the System V signals SIGEMT, SIGBUS, and SIGIOT are implementation dependent and are not listed above.) Specific implementations may have other implementation-specific signals.

*Func* is assigned one of three values: SIG\_DFL, SIG\_IGN, or a *function address*. The actions prescribed by these values are as follows:

**SIG\_DFL** terminate process upon receipt of a signal.  
Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). In addition if *sig* is one of the signals shown with a † above, implementation-dependent abnormal process termination routines such as a core dump, may be invoked.

**SIG\_IGN** ignore signal.  
The signal *sig* is to be ignored.

**Note:** the signal SIGKILL cannot be ignored.

*function address* — catch signal  
Upon receipt of the signal *sig*, the receiving process is to execute the



signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Additional arguments may be passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of *func* for the caught signal will be set to `SIG_DFL` unless the signal is `SIGILL` or `SIGTRAP`.

*Signal* will not catch an invalid function argument, *func*, and results are undefined when an attempt is made to execute the function at the bad address.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted, except for implementation defined signals where this may not be true.

When a signal that is to be caught occurs during a `read(2)`, a `write(2)`, an `open(2)`, or an `ioctl(2)` system call on a slow device (like a terminal; but not a file), during a `pause(2)` system call, or during a `wait(2)` system call that does not return immediately, the signal catching function will be executed and then the interrupted system call may return a -1 to the calling process with *errno* set to `[EINTR]`.

**Note:** The signal `SIGKILL` cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending `SIGKILL` signal.

#### ERRORS

`[EINVAL]` *Signal* will fail if *sig* is an illegal signal number, or is `SIGKILL`.

#### RETURN VALUE

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of `(int (*)( ))-1` is returned and *errno* is set to indicate the error.

#### SEE ALSO

`exit(2)`, `ioctl(2)`, `kill(2)`, `open(2)`, `pause(2)`, `wait(2)`, `read(2)`, `write(2)`, `setjmp(3C)`, `signal(5)`.

#### APPLICATIONS USAGE

For portability, applications should use only the symbolic names of signals rather than their values and use only the set of signals defined here. Specific implementations may have additional signals.

The signal `SIGSEGV` is only included for compatibility with existing applications. It is not part of the SVID and should not be used in programs that wish to be portable.

If signals are being caught, then *errno* may be changed by errors that occur in the signal-catching function. Its value cannot be relied upon if there is any possibility of a signal arriving between the setting of *errno* and its use.

#### FUTURE DIRECTIONS

A macro `SIG_ERR` will be defined in `<signal.h>` to represent the return value `(int (*)( ))-1` by *signal* in the case of error.

**CHANGE HISTORY**

**Issue 1**

Derived from the entry in Issue 1 of the SVID with the inclusion of the two extra signals, SIGABRT and SIGSEGV. (Note that the addition of SIGABRT is forecast in the FUTURE DIRECTIONS section of the SVID.)





## NAME

stat, fstat — get file status

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

## DESCRIPTION

*Path* points to a path name naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *Stat* obtains information about the named file.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

*Buf* is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
ushort  st_mode;    /* File mode, see mknod(2) */
ino_t   st_ino;     /* Inode number */
dev_t   st_dev;     /* ID of device containing a */
          /* directory entry for this file */
dev_t   st_rdev;    /* ID of device */
          /* This entry is defined only for character */
          /* special or block special files */
short   st_nlink;   /* Number of links */
ushort  st_uid;     /* User ID of the file's owner */
ushort  st_gid;     /* Group ID of the file's group */
off_t   st_size;    /* File size in bytes */
time_t  st_atime;   /* Time of last access */
time_t  st_mtime;   /* Time data last modified */
time_t  st_ctime;   /* Time of last file status change */
          /* Times in seconds since */
          /* 00:00:00 GMT, January 1, 1970 */
```

*st\_atime* Time when file data was last accessed. Changed by the following system calls: *creat*(2), *fcntl*(2), *mknod*(2), *pipe*(2), *read*(2), and *utime*(2).

*st\_mtime* Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).

*st\_ctime* Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

## ERRORS

*Stat* will fail if one or more of the following are true:

- [ENOTDIR] A component of the path prefix is not a directory.
- [ENOENT] The named file does not exist.
- [EACCES] Search permission is denied for a component of the path prefix.
- [EFAULT] *Buf* or *path* points to an invalid address.

*Fstat* will fail if one or more of the following are true:

- [EBADF] *Fildes* is not a valid open file descriptor.
- [EFAULT] *Buf* points to an invalid address.

#### RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

#### SEE ALSO

*chmod(2)*, *chown(2)*, *creat(2)*, *dup(2)*, *fcntl(2)*, *link(2)*, *mknod(2)*, *open(2)*, *pipe(2)*, *read(2)*, *time(2)*, *unlink(2)*, *utime(2)*, *write(2)*, *types(5)*, *stat(5)*.

#### CHANGE HISTORY

##### Issue 1

Derived from the entry in Issue 1 of the SVID.

## NAME

stime — set time

## SYNOPSIS

```
int stime (tp)
long *tp;
```

## DESCRIPTION

*Stime* sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

## ERRORS

*Stime* will fail if the following is true:

[EPERM]           the effective user ID of the calling process is not super-user, or it is not possible to change the time.

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

time(2).

## APPLICATION USAGE

On some systems, it may not be possible or desirable to alter the time from a UNIX process.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

The text "", or it is not possible to change the time" has been added to the [EPERM] error.





## NAME

sync — update super-block

## SYNOPSIS

```
void sync ( )
```

## DESCRIPTION

*Sync* causes all information in kernel buffers that updates a file system to be written out to the file system. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

The term "kernel buffers" refers to system buffers which are invisible to the user and are only present to improve performance. It does *not* mean buffering provided by the *stdio*(3S) functions.

## SEE ALSO

*stdio*(3S).

## APPLICATION USAGE

Application programs are not expected to use *sync*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following changes:

In the first line of the DESCRIPTION section, the SVID has the term "transient memory" in place of "kernel buffers". The X/OPEN definition has included the last paragraph of the DESCRIPTION section to achieve the same purpose.

Explanatory text has been added to the final paragraph of the DESCRIPTION section.





## NAME

*time* — get time

## SYNOPSIS

long *time* (*tloc*)

long \**tloc*;

## DESCRIPTION

*Time* returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

As long as *tloc* is not a NULL pointer, the return value is also stored in the location to which *tloc* points.

## ERRORS

[EFAULT] *tloc* points to an invalid address.

## RETURN VALUE

Upon successful completion, *time* returns the value of time. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## APPLICATION USAGE

Some implementations of *time* fail to check the validity of *tloc* and give undefined behaviour if *tloc* points to an invalid address.

## SEE ALSO

*stime*(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following changes:

The warning above in **APPLICATION USAGE** is implicit in the SVID definition, which assumes that no systems check *tloc*. The SVID does not mention [EFAULT] or *errno*.

## Issue 2

A line has been removed from the SYNOPSIS section, as the functionality is now described in the DESCRIPTION.



## NAME

*times* — get process and child elapsed process times

## SYNOPSIS

```
#include <sys/types.h>
```

```
#include <sys/times.h>
```

```
long times (buffer)
```

```
struct tms *buffer;
```

## DESCRIPTION

*Times* fills the structure pointed to by *buffer* with time-accounting information. The structure *tms*, which is defined in `<sys/times.h>`, contains the following elements:

```
time_t  tms_utime;
time_t  tms_stime;
time_t  tms_cutime;
time_t  tms_cstime;
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait(2)*. All times are defined in {CLK\_TCK}ths of a second.

*Tms\_utime* is the CPU time used while executing instructions in the user space of the calling process.

*Tms\_stime* is the CPU time used by the system on behalf of the calling process.

*Tms\_cutime* is the sum of the *tms\_utimes* and *tms\_cutimes* of the child processes.

*Tms\_cstime* is the sum of the *tms\_stimes* and *tms\_cstimes* of the child processes.

## ERRORS

[EFAULT] *Times* will fail if *buffer* points to an illegal address.

## RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in {CLK\_TCK}ths of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*exec(2)*, *fork(2)*, *time(2)*, *wait(2)*, *types(5)* *times(5)*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

ulimit — get and set user limits

## SYNOPSIS

```
long ulimit (cmd, newlimit)
int cmd;
long newlimit;
```

## DESCRIPTION

*Ulimit* provides for control over process limits. The *cmd* values available are:

- 1 Get the file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.
- 2 Set the file size limit of the process to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit.
- 3 Get the maximum possible *brk* value see *brk(2)*. *Brk(2)* is optional.

## ERRORS

[EPERM] *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit.

## RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*brk(2)*, *write(2)*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The *cmd* value 3 case has been added. This arises because *brk(2)* is included as an *optional* facility by X/OPEN, although it is omitted from the SVID.





NAME

umask — set and get file creation mask

SYNOPSIS

```
int umask (cmask)
int cmask;
```

DESCRIPTION

*Umask* sets the process's file mode creation mask (see *creat(2)*) to *cmask* and returns the previous value of the mask. Only the owner, group, other permission bits of *cmask* and the file mode creation mask are used.

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

chmod(2), creat(2), mknod(2), open(2).

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID.



## NAME

umount — unmount a file system

## SYNOPSIS

```
int umount (spec)
char *spec;
```

## DESCRIPTION

*Umount* requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

*Umount* may only be invoked with an effective user ID equal to super-user.

## ERRORS

*Umount* will fail if one or more of the following are true:

[EPERM]	The process's effective user ID is not super-user.
[ENXIO]	The device associated with <i>spec</i> does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[ENOTBLK]	<i>Spec</i> is not a block special device.
[EINVAL]	<i>Spec</i> is not mounted.
[EBUSY]	A file on <i>spec</i> is busy.
[EFAULT]	<i>Spec</i> points to an illegal address.

## RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

mount(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

In the last paragraph of the **DESCRIPTION** section the SVID reads: "*Umount* may be invoked only by the super-user."

## Issue 2

The word "only" has been added to the last paragraph of the **DESCRIPTION**.





## NAME

uname — get name of current system

## SYNOPSIS

```
#include <sys/utsname.h>
```

```
int uname (name)
```

```
struct utsname *name;
```

## DESCRIPTION

*Uname* stores information identifying the current system in the structure pointed to by *name*.

*Uname* uses the structure defined in `<sys/utsname.h>` whose members are:

```
char    sysname[SYS_NMLN];
char    nodename[SYS_NMLN];
char    release[SYS_NMLN];
char    version[SYS_NMLN];
char    machine[SYS_NMLN];
```

*Uname* returns a null-terminated character string naming the current system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Machine* contains a standard name that identifies the hardware that the system is running on.

## ERRORS

[EFAULT] *Uname* will fail if *name* points to an invalid address.

## RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

utsname(5).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The term "UNIX system" has been changed to "system".





## NAME

unlink — remove directory entry

## SYNOPSIS

```
int unlink (path)
char *path;
```

## DESCRIPTION

*Unlink* removes the directory entry named by the path name pointed to by *path*. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, space occupied by the file is *not* released until all references to the file have been closed.

## ERRORS

The named file is unlinked unless one or more of the following are true:

- |           |  |
|-----------|--|
| [ENOTDIR] | A component of the path prefix is not a directory.   |
| [ENOENT]  | The named file does not exist.   |
| [EACCES]  | Search permission is denied for a component of the path prefix.  |
| [EACCES]  | Write permission is denied on the directory containing the link to be removed.                           |
| [EPERM]   | The named file is a directory and the effective user ID of the process is not super-user.                |
| [EBUSY]   | The entry to be unlinked is the mount point for a mounted file system.                                   |
| [ETXTBSY] | The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. |
| [EROFS]   | The directory entry to be unlinked is part of a read-only file system.                                   |
| [EFAULT]  | <i>Path</i> points outside the process's allocated address space.  |

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

close(2), link(2), open(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The last sentence of the DESCRIPTION section has been clarified. The SVID reads: "...removed, the removal is postponed until all references to the file have been closed".



## NAME

ustat — get file system statistics

## SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>
```

```
int ustat (dev, buf)
int dev;
struct ustat *buf;
```

## DESCRIPTION

*Ustat* returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure that includes the following elements:

daddr_t	f_tfree;	/* Total free blocks */
ino_t	f_tinode;	/* Number of free inodes */
char	f_fname[6];	/* Filsys name or NULL */
char	f_fpack[6];	/* Filsys pack name or NULL */

The last two fields, *f\_fname* and *f\_fpack*, may not have meaningful information on all systems, and, in that case, will contain the NULL character.

## ERRORS

*Ustat* will fail if one or more of the following are true:

[EINVAL]	<i>Dev</i> is not the device number of a device containing a mounted file system.
[EFAULT]	<i>Buf</i> points outside the process's allocated address space.

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

stat(2), types(5), ustat(5).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.



1. The first part of the document is a list of names and addresses.

## NAME

*utime* — set file access and modification times

## SYNOPSIS

```
#include <sys/types.h>

int utime (path, times)
char *path;
struct utimbuf *times;
```

## DESCRIPTION

*Path* points to a path name naming a file. *Utime* sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the superuser may use *utime* this way.

The times in the following structure *utimbuf* are measured in seconds since 00:00:00 GMT, January 1, 1970:

```
struct utimbuf{
    time_t  actime;
    time_t  modtime;
};
```

*Utime* will also cause the time of the last file status change (*st\_ctime*) to be updated, see *stat*(5).

## ERRORS

*Utime* will fail if one or more of the following are true:

- |           |  |
|-----------|--|
| [ENOENT]  | The named file does not exist.   |
| [ENOTDIR] | A component of the path prefix is not a directory.   |
| [EACCES]  | Search permission is denied by a component of the path prefix.   |
| [EPERM]   | The effective user ID is not super-user and not the owner of the file and <i>times</i> is not NULL.                        |
| [EACCES]  | The effective user ID is not super-user and not the owner of the file and <i>times</i> is NULL and write access is denied. |
| [EROFS]   | The file system containing the file is mounted read-only.  |
| [EFAULT]  | <i>Times</i> is not NULL and points outside the process's allocated address space.   |
| [EFAULT]  | <i>Path</i> points outside the process's allocated address space.  |

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*stat*(2), *types*(5).

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID, except that the SVID does not give an explicit declaration of *struct utimbuf*.

#### Issue 2

Aligned with the entry in Issue 2 of the SVID which includes the declaration of *struct utimbuf*.



## NAME

wait — wait for child process to stop or terminate

## SYNOPSIS

```
int wait (stat_loc)
int *stat_loc;
```

## DESCRIPTION

*Wait* suspends the calling process until one of the immediate children terminates or until a child that is being traced stops because it has hit a break point. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat\_loc* is not a NULL pointer, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by *stat\_loc*. *Status* can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

If the child process stopped, the low order 8 bits of status will be set to 0177 and the next 8 bits of status will contain the number of the signal that caused the process to stop.

If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the next 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit(2)*.

If the child process terminated due to a signal, the first 7 bits of the low order 8 bits will contain the number of the signal that caused the termination and the next 8 bits of status will be zero. In addition, if abnormal process termination routines (see *signal(2)*) were successfully completed, then the low order eighth bit (i.e., bit 0200) will be set.

If a parent process terminates without waiting for its child processes to terminate, a special system process inherits the child processes, see *exit(2)*.

*Wait* will fail and its actions are undefined if *stat\_loc* points to an illegal address.

## ERRORS

*Wait* will fail and return immediately if the following is true:

[ECHILD]           The calling process has no existing unwaited-for child processes.

## RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of -1 is returned to the calling process and *errno* is set to [EINTR]. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*exec(2)*, *exit(2)*, *fork(2)*, *pause(2)*, *signal(2)*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

The number of bits which contain the signal number has been corrected from 8 to 7.





## NAME

write — write on a file

## SYNOPSIS

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

## DESCRIPTION

*Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

*Write* attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the *O\_APPEND* flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit*, see *ulimit* (2), or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If a *write* to a pipe (or fifo) of {PIPE\_BUF} bytes or less is requested and less than *nbytes* bytes of free space is available in the pipe, one of the following will occur:

If the *O\_NDELAY* flag is clear, the process will block until at least *nbytes* of space is available in the pipe and then the *write* will take place, or

If the *O\_NDELAY* flag is set, the process will not block and the function *write* will return 0.

A *write* request of greater than {PIPE\_BUF} bytes to a pipe (or fifo) will behave differently:

If the *O\_NDELAY* flag is clear, the process will block if the pipe is full. As space becomes available in the pipe, the data from the *write* request will be written piecemeal — in multiple smaller amounts until the request is fulfilled. Thus, data from a *write* request of more than {PIPE\_BUF} bytes may be interleaved on arbitrary byte boundaries with data written by other processes.

If the *O\_NDELAY* flag is set and the pipe is full, the process will not block and the function *write* will return 0.

If the *O\_NDELAY* flag is set and the pipe is not full, the process will not block and as much data as will currently fit on the pipe will be written and the function *write* will return the number of bytes written. In this case, only part of the data are written, but what data are written will not be interleaved with data from other processes.



In contrast to *write* requests of more than {PIPE\_BUF} bytes, data from a *write* request of {PIPE\_BUF} bytes or less will never be interleaved in the pipe with data from other processes.

### ERRORS

*Write* will fail and the file pointer will remain unchanged if one or more of the following are true:

- [EBADF] *Fildes* is not a valid file descriptor open for writing.
- [EPIPE] and SIGPIPE signal  
An attempt is made to write to a pipe that is not open for reading by any process.
- [EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size, {FCHR\_MAX}. See *ulimit(2)*.
- [EINTR] A signal was caught during the *write* system call.
- [EFAULT] *Buf* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.
- [ENOSPC] There is no free space remaining on the device containing the file.
- [EIO] An I/O error occurred on a special file.
- [ENXIO] A request was made of a non-existent special file, or the request was outside the capabilities of the device.

### RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

### SEE ALSO

*creat(2)*, *dup(2)*, *fcntl(2)*, *lseek(2)*, *open(2)*, *pipe(2)*, *ulimit(2)*.

### APPLICATION USAGE

Normally, applications should use *stdio(3S)* library routines to open, close, read and write files. Thus, if an application had used the *stdio* routine *fopen(3S)* to open a file, it would use the *stdio* routine *fwrite(3S)* rather than *write(2)*.

Warning: *write* errors to I/O devices give implementation defined results.

### FUTURE DIRECTIONS

[EAGAIN] will be returned in *errno* if the no delay mode is in use on the file and the process will be delayed in the write operation.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The [EIO] and [ENXIO] errors have been added.

Issue 2

The **FUTURE DIRECTION** on mandatory or enforcement mode file and record locking has been dropped.

Aligned with the entry in Issue 2 of the SVID by applying the following changes:

The paragraphs describing the operation of pipes have been added.

The text ", {FCHR\_MAX}" has been added to the description of the [EFBIG] error.

The SVID now includes descriptions of the [EIO] and [ENXIO] errors.





## ***Subroutines and Libraries***

This chapter describes functions conventionally found in various libraries, other than those functions that directly invoke X/OPEN system calls, which are described in Chapter 2 of this part of the Guide. Certain major collections are identified by a letter after the section number. The established 3C, 3M, 3S, and 3X nomenclature is preserved, but is not important in the context of an interface definition.

- (3C) These functions, together with those of Chapter 2 and those marked (3S), constitute the conventional C Library. Declarations for some of these functions may be obtained from `#include` files indicated on the appropriate pages.
- (3M) These functions constitute the *optional* math group. Declarations for these functions may be obtained from the `#include` file `<math.h>`.
- (3S) These functions constitute the "standard I/O group" (see `stdio(3S)`). Declarations for these functions may be obtained from the `#include` file `<stdio.h>`.
- (3X) Various specialised functions.

Functions in the math group (3M) may return the conventional values 0 or HUGE (the largest single-precision floating-point number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable `errno` is set to the value [EDOM] or [ERANGE].



NAME

abort — generate an abnormal process abort

SYNOPSIS

int abort ()

DESCRIPTION

*Abort* first closes all open files, if possible, then causes the process abort signal, **SIGABRT**, to be sent to the process. This invokes abnormal process termination routines, such as a *core dump*, which are implementation dependent.

ERRORS

None.

APPLICATION USAGE

**SIGABRT** is not intended to be caught.

SEE ALSO

exit(2), signal(2), kill(2).

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The signal is identified as **SIGABRT**. The SVID does not identify the signal, but forecasts in the FUTURE DIRECTIONS section that it will be **SIGABRT**. (In UNIX System V Release 2.0 it is **SIGIOT**.)





## NAME

abs — return integer absolute value

## SYNOPSIS

```
int abs (i)
int i;
```

## DESCRIPTION

*Abs* returns the absolute value of its integer operand.

## SEE ALSO

*fabs*(3M).

## APPLICATION USAGE

In two's-complement representation, the absolute value of the negative integer with largest magnitude {INT\_MIN} is undefined. Some implementations trap this error, but others simply ignore it.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

assert — verify program assertion

## SYNOPSIS

```
#include <assert.h>

void assert (expression)
int expression;
```

## DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

## SEE ALSO

abort(3C).

## APPLICATION USAGE

Forcing a definition of the name *NDEBUG*, either from the compiler command line or with the preprocessor control statement *#define NDEBUG* ahead of the *#include <assert.h>* statement, will stop assertions from being compiled into the program.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.



## NAME

*j0*, *j1*, *jn*, *y0*, *y1*, *yn* — Bessel functions (OPTIONAL)

## SYNOPSIS

```
#include <math.h>
```

```
double j0 (x)
```

```
double x;
```

```
double j1 (x)
```

```
double x;
```

```
double jn (n, x)
```

```
int n;
```

```
double x;
```

```
double y0 (x)
```

```
double x;
```

```
double y1 (x)
```

```
double x;
```

```
double yn (n, x)
```

```
int n;
```

```
double x;
```

## DESCRIPTION

*J0* and *J1* return Bessel functions of *x* of the first kind of orders 0 and 1 respectively. *Jn* returns the Bessel function of *x* of the first kind of order *n*.

*Y0* and *Y1* return Bessel functions of *x* of the second kind of orders 0 and 1 respectively. *Yn* returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

## RETURN VALUE

Non-positive arguments cause *y0*, *y1* and *yn* to return the value -HUGE and to set *errno* to [EDOM]. In addition, a message indicating DOMAIN error is printed on the standard error output.

Arguments too large in magnitude cause *j0*, *j1*, *y0* and *y1* to return zero and to set *errno* to [ERANGE]. In addition, a message indicating TLOSS error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

## SEE ALSO

*matherr*(3M).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID. The *optional* mathematical group is mandatory in the SVID.





## NAME

bsearch — binary search a sorted table

## SYNOPSIS

```
#include <search.h>

char *bsearch (key, base, nel, width, compar)
char *key;
char *base;
unsigned nel, width;
int (*compar)();
```

## DESCRIPTION

*Bsearch* is a binary search routine generalised from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function, *compar*. *Key* points to a datum instance to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Width* is the size of an element in bytes. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero as accordingly the first argument is to be considered less than, equal to, or greater than the second.

## RETURN VALUE

A NULL pointer is returned if the key cannot be found in the table.

## SEE ALSO

hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C), search(5).

## APPLICATION USAGE

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

## EXAMPLE

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>
#include <search.h>

#define TABSIZE 1000

struct node {
    char *string;
    int length;
};

struct node table[TABSIZE]; /* table to be searched */
```

```

{
    struct node *node_ptr, node;
    int node_compare( ); /* routine to compare 2 nodes */
    char str_space[20]; /* space to read string into */

    node.string = str_space;
    while (scanf("%s", node.string) != EOF) {
        node_ptr = (struct node *)bsearch((char *)&node,
            (char *)table, TABSIZE,
            sizeof(struct node), node_compare);
        if (node_ptr != NULL) {
            (void)printf("string = %20s, length = %d\n",
                node_ptr->string, node_ptr->length);
        } else {
            (void)printf("not found: %s\n", node.string);
        }
    }
}
/*
    This routine compares two nodes based on an
    alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
char *node1, *node2;
{
    return strcmp((struct node *) node1->string,
        (struct node *) node2->string);
}

```

**CHANGE HISTORY****Issue 1**

Derived from the entry in Issue 1 of the SVID with the following change:

A programming error in the *node\_compare* part of the example has been corrected.



## NAME

`clock` — report CPU time used

## SYNOPSIS

`long clock ( )`

## DESCRIPTION

*Clock* returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait(2)* or *system(3S)*.

## SEE ALSO

*times(2)*, *wait(2)*, *system(3S)*.

## APPLICATION USAGE

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.



## NAME

*toupper*, *tolower*, *\_toupper*, *\_tolower*, *toascii* — translate characters (NLS)

## SYNOPSIS

```
#include <ctype.h>
```

```
int toupper (c)
```

```
int c;
```

```
int tolower (c)
```

```
int c;
```

```
int _toupper (c)
```

```
int c;
```

```
int _tolower (c)
```

```
int c;
```

```
int toascii (c)
```

```
int c;
```

## DESCRIPTION

*Toupper* and *tolower* have as domain the range of *getc*(3S): the integers from —1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros *\_toupper* and *\_tolower* are macros that accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. The macro *\_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. The macro *\_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

*Toascii* yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

## SEE ALSO

*ctype*(3C), *getc*(3S).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

*crypt*, *setkey*, *encrypt* — generate DES encryption

## SYNOPSIS

```
char *crypt (key, salt)
char *key, *salt;

void setkey (key)
char *key;

void encrypt (block, edflag)
char *block;
int edflag;
```

## DESCRIPTION

*Crypt* is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

*Key* is a user's typed password. *Salt* is a two-character string chosen from the set [a-zA-Z0-9./]; this string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the *salt* itself.

The *setkey* and *encrypt* entry provides (rather primitive) access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the above mentioned algorithm to encrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value of 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is zero (0), the argument is encrypted.

## APPLICATION USAGE

The return value of *crypt* points to static data that are overwritten by each call.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

*ctermid* — generate file name for terminal

## SYNOPSIS

```
#include <stdio.h>
```

```
char *ctermid (s)
```

```
char *s;
```

## DESCRIPTION

*Ctermid* generates the path name of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, with contents overwritten at the next call to *ctermid*, and returned address. Otherwise, *s* is assumed to point to a character array of at least {L\_ctermid} elements; the path name is placed in this array and the value of *s* is returned. The constant {L\_ctermid} is defined in the <stdio.h> header file.

## SEE ALSO

*ttyname*(3C).

## APPLICATION USAGE

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (such as */dev/tty*) that will refer to the terminal if used as a file name. Thus *ttyname* is useful only if the process already has at least one file open to a terminal.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The words "such as" have been inserted before *"/dev/tty"* in the APPLICATION USAGE section.



## NAME

`ctime`, `localtime`, `gmtime`, `asctime`, `tzset`, `timezone`, `daylight`, `tzname` — convert date and time to string (NLS)

## SYNOPSIS

```
#include <time.h>

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone;
extern int daylight;
extern char *tzname[2];
void tzset ( )
```

## DESCRIPTION

*Ctime* converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form:

Sun Sep 16 01:03:52 1973 \n \0

All the fields have constant width. *Localtime* and *gmtime* return pointers to *tm* structures, described below. *Localtime* corrects for the time zone and possible Daylight Saving Time; *gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the system uses.

*Asctime* converts a *tm* structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Declarations of all the functions and externals, and the *tm* structure, are in the `<time.h>` header file. The structure contains the following fields:

```
int tm_sec;      /* seconds (0 - 59) */
int tm_min;      /* minutes (0 - 59) */
int tm_hour;     /* hours (0 - 23) */
int tm_mday;     /* day of month (1 - 31) */
int tm_mon;      /* month of year (0 - 11) */
int tm_year;     /* year - 1900 */
int tm_wday;     /* day of week (Sunday is 0) */
int tm_yday;     /* day of year (0 - 365) */
int tm_isdst;    /* Daylight savings time flag */
```

*Tm\_isdst* is non-zero if Daylight Saving Time is in effect.

The external `long` variable *timezone* contains the difference, in seconds, between GMT and local standard time (in MET, *timezone* is  $-1 \times 60 \times 60$ ); the external variable *daylight* is non-zero if and only if some Daylight Saving Time conversion should be applied.



If an environment variable named TZ is present, *asctime* uses the contents of the variable to override the default time zone. The value of TZ must be a three-letter time zone name, followed by an optional minus sign (for zones east of Greenwich) and a series of digits representing the difference between local time and GMT in hours; this is followed by an optional three letter name for a daylight time zone. The setting for most of continental Europe would be MET-1EET. The effects of setting TZ are thus to change the values of the external variables *timezone* and *daylight*. In addition, the time zone names contained in the external variable

```
char *tzname[2] = {"MET", "EET"};
```

are set from the environment variable TZ. The function *tzset* sets these external variables from TZ; *tzset* is called by *asctime* and may also be called explicitly by the user.

Note that Daylight Saving Time conversion is applied all year round if there is a daylight time zone name present in the TZ variable. I.e., *daylight* is non-zero all year round, but *tm\_isdst* is non-zero only when Daylight Saving Time is in effect.

#### SEE ALSO

*time*(2), *getenv*(3C), *environ*(5).

#### APPLICATION USAGE

The return values point to static data whose content is overwritten by each call.

The usefulness of these functions is diminished by the fact that they are inappropriate in those parts of the world which do not base their reckoning of time upon GMT. The algorithm used to determine whether Daylight Saving Time applies depends on the location in question; as it is usually supplied, the implementation of *ctime* only knows about a small number of the necessary conversions. There is no agreed international standard for timezone names. The following names are suggested for Europe:

WET	(GMT)	Western European Time, e.g., U.K.
MET	(GMT+1)	Middle European Time
EET	(GMT+2)	Eastern European Time

#### FUTURE DIRECTIONS

The argument *clock* to *ctime*, *localtime* and *gmtime* will be declared through the *typedef* facility as a pointer to *time\_t*.

The number in TZ will be defined as an optional minus sign followed by two hour digits and two minute digits, *[-]hhmm*, to represent fractional timezones.

#### CHANGE HISTORY

##### Issue 1

Derived from the entry in Issue 1 of the SVID with the following changes:

European timezone names have replaced references to North America, and the names *timezone*, *daylight* and *tzname* have been added to the **NAME** section.

##### Issue 2

The last paragraph of the **DESCRIPTION** section has been added to clarify the use of the *daylight* and *tm\_isdst* variables.

In the **SYNOPSIS** section, the call for the inclusion of `<sys/types.h>` has been removed to reflect a correction made in Issue 2 of the SVID.

## NAME

*isalpha*, *isupper*, *islower*, *isdigit*, *isxdigit*, *isalnum*, *isspace*, *ispunct*, *isprint*, *isgraph*, *isctrl*, *iscntrl*, *isascii* — classify characters (NLS)

## SYNOPSIS

```
#include <ctype.h>
```

```
int isalpha (c)
```

```
int c;
```

```
...
```

## DESCRIPTION

These macros classify character-coded integer values. Each is a predicate returning non-zero for true, zero (0) for false. *isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value EOF, see *stdio*(3S).

*isalpha* *c* is a letter.

*isupper* *c* is an upper-case letter.

*islower* *c* is a lower-case letter.

*isdigit* *c* is a digit [0-9].

*isxdigit* *c* is a hexadecimal digit [0-9], [A-F] or [a-f].

*isalnum* *c* is an alphanumeric (letter or digit).

*isspace* *c* is a space, tab, carriage return, new-line, vertical tab or form-feed.

*ispunct* *c* is a punctuation character (neither control nor alphanumeric).

*isprint* *c* is a printing character, code 040 (space) through 0176 (tilde).

*isgraph* *c* is a printing character, like *isprint* except false for space.

*isctrl* *c* is a delete character (0177) or an ordinary control character (less than 040).

*isascii* *c* is an ASCII character, code between 0 and 0177 inclusive.

## RETURN VALUE

If the argument to any of these macros is not in the domain of the function, the result is undefined.

## APPLICATIONS USAGE

To ensure applications portability, especially across natural languages, no other character classification method should be used.

## SEE ALSO

*stdio*(3S), *ctype*(5).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

The reference to *ctype*(5) has been added to the SEE ALSO section.





## NAME

*cuserid* — character login name of the user

## SYNOPSIS

```
#include <stdio.h>
```

```
char *cuserid (s)
```

```
char *s;
```

## DESCRIPTION

*Cuserid* generates a character representation of the login name of the owner of the current process. If *s* is NULL, this representation is generated in an internal static area, the address of which is returned. If *s* is not NULL, *s* is assumed to point to an array of at least {L\_ *cuserid*} characters; the representation is left in this array. The symbolic constant {L\_ *cuserid*} is defined in <stdio.h>.

## RETURN VALUE

If *s* is NULL and the login name cannot be found, *cuserid* returns NULL; if *s* is not NULL and the login name cannot be found, the null character “\0” will be placed at \**s*.

## SEE ALSO

getlogin(3C), getpwent(3C).

## APPLICATION USAGE

*Cuserid* uses *getpwnam*(3C); thus the results of a user's call to *getpwnam*(3C) will be obliterated by a subsequent call to *cuserid*.

## CHANGE HISTORY

## Issue 1

This function is not included in the SVID. It comes from UNIX System V Release 2.0.



## NAME

*closedir*, *opendir*, *readdir*, *rewinddir*, *seekdir*, *telldir* — directory operations

## SYNOPSIS

```
#include <sys/types.h>
#include <limits.h>
#include <sys/dirent.h>

void closedir(dirp)
DIR *dirp;

DIR *opendir(filename)
char *filename;

struct dirent *readdir(dirp)
DIR *dirp;

void rewinddir(dirp)
DIR *dirp;

void seekdir(dirp, loc)
DIR *dirp;
long loc;

long telldir(dirp)
DIR *dirp;
```

## DESCRIPTION

The function *closedir* closes the directory-descriptor indicated by the argument *dirp* and frees the DIR structure associated with the directory-descriptor.

The function *opendir* opens the directory named by the argument *filename* and returns a pointer to the DIR structure associated with the directory.

The function *readdir* returns a pointer to the next non-empty directory entry in the directory structure specified by the argument *dirp*. The structure *dirent* defined by the *<sys/dirent.h>* header file describes a directory entry.

The function *rewinddir*(*dirp*) is equivalent to the following:

```
seekdir(dirp, 0L)
```

The function *seekdir* sets the position of the next *readdir* operation on the directory specified by the argument *dirp*. The argument *loc* is obtained from a *telldir* operation. The new position reverts to the position that was associated with the directory when the *telldir* operation was performed.

The function *telldir* returns the current location of the directory associated with the argument *dirp*. Values returned by *telldir* are valid only if the directory has not changed due to compaction or expansion.



### EXAMPLE

The following sample code will search a directory for the entry *name*:

```
dirp = opendir(".");
while ((dp = readdir(dirp)) != NULL)
    if (strcmp(dp->d_name, name) == 0) {
        closedir(dirp);
        return FOUND;
    }
closedir(dirp);
return NOT_FOUND;
```

### RETURN VALUE

The function *opendir* returns a NULL pointer if *filename* cannot be accessed, or if *filename* is not a directory, or if enough memory to hold a DIR structure or a buffer for the directory entries cannot be allocated.

The function *readdir* returns a NULL pointer upon reaching the end of the directory or upon detecting an invalid location in the directory.

### SEE ALSO

*open(2)*, *close(2)*, *read(2)*, *lseek(2)*, *dirent(5)*.

### APPLICATION USAGE

This is the recommended portable way to examine the contents of a directory.

### CHANGE HISTORY

First released in Issue 2.

### Issue 2

This function was added in Issue 2.

This is not in the SVID. It is specific to X/OPEN systems.

## NAME

*drand48*, *erand48*, *lrand48*, *nrand48*, *mrand48*, *jrand48*, *srand48*, *seed48*, *lcong48*  
— generate uniformly distributed pseudo-random numbers

## SYNOPSIS

```
double drand48 ( )
double erand48 (xsubi)
unsigned short xsubi[3];
long lrand48 ( )
long nrand48 (xsubi)
unsigned short xsubi[3];
long mrand48 ( )
long jrand48 (xsubi)
unsigned short xsubi[3];
void srand48 (seedval)
long seedval;
unsigned short *seed48 (seed16v)
unsigned short seed16v[3];
void lcong48 (param)
unsigned short param[7];
```

## DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

N.B. In the following, the formal mathematical notation  $[0.0, 1.0)$  indicates an interval including 0.0 but not including 1.0.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval  $[0.0, 1.0)$ .

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval  $[0, 2^{31})$ .

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval  $[-2^{31}, 2^{31})$ .

Functions *srand48*, *seed48* and *lcong48* are initialisation entry points, one of which should be invoked before either *drand48*, *lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initialiser values will be supplied automatically if *drand48*, *lrand48* or *mrand48* is called without a prior call to an initialisation entry point.) Functions *erand48*, *nrand48* and *jrand48* do not require an initialisation entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values,  $X_i$ , according to the linear congruential formula

$$X_{n+1} = (aX_n + c) \bmod m \quad n > 0.25n' = 0.$$



The parameter  $m = 2^{48}$ ; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value  $a$  and the addend value  $c$  are given by

$$\begin{aligned} a &= 5DEECE66D_{16} = 273673163155_8 \\ c &= B_{16} = 13_8. \end{aligned}$$

The value returned by any of the functions *drand48*, *erand48*, *lrand48*, *nrand48*, *mrnd48* or *jrand48* is computed by first generating the next 48-bit  $X_i$  in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of  $X_i$  and transformed into the returned value.

The functions *drand48*, *lrand48* and *mrnd48* store the last 48-bit  $X_i$  generated in an internal buffer; that is why they must be initialised prior to being invoked. The functions *erand48*, *nrand48* and *jrand48* require the calling program to provide storage for the successive  $X_i$  values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialised; the calling program merely has to place the desired initial value of  $X_i$  into the array and pass it as an argument. By using different arguments, functions *erand48*, *nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e. the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initialiser function *srand48* sets the high-order 32 bits of  $X_i$  to the {LONG\_BIT} bits contained in its argument. The low-order 16 bits of  $X_i$  are set to the arbitrary value 330E<sub>16</sub>.

The initialiser function *seed48* sets the value of  $X_i$  to the 48-bit value specified in the argument array. In addition, the previous value of  $X_i$  is copied into a 48-bit internal buffer, used only by *seed48*, and a pointer to this buffer is the value returned by *seed48*. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last  $X_i$  value, and then use this value to reinitialise via *seed48* when the program is restarted.

The initialisation function *lcong48* allows the user to specify the initial  $X_i$ , the multiplier value  $a$ , and the addend value  $c$ . Argument array elements *param*[0-2] specify  $X_i$ , *param*[3-5] specify the multiplier  $a$ , and *param*[6] specifies the 16-bit addend  $c$ . After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values,  $a$  and  $c$ , specified on the previous page.

#### SEE ALSO

rand(3C).

#### CHANGE HISTORY

##### Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The SVID paragraph in the APPLICATIONS USAGE section has been moved up into the DESCRIPTION section and added onto the end of the *seed48* paragraph.



## NAME

*ecvt*, *fcvt*, *gcvt* — convert floating-point number to string (NLS)

## SYNOPSIS

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

## DESCRIPTION

*Ecvt* converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero (0). The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

*Fcvt* is identical to *ecvt*, except that the correct digit has been rounded for *printf* "%f" (FORTRAN F-format) output of the number of digits specified by *ndigit*.

*Gcvt* converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

## SEE ALSO

*printf*(3S).

## BUGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

## CHANGE HISTORY

## Issue 1

This function is not included in the SVID. It comes from UNIX System V Release 2.0.



## NAME

*end*, *etext*, *edata* — last locations in program (OPTIONAL)

## SYNOPSIS

```
extern end;  
extern etext;  
extern edata;
```

## DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialised data region, and *end* above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk(2)*, *malloc(3X)*, standard input/output (*stdio(3S)*), and so on. Thus, the current value of the program break should be determined by *sbrk(0)*, see *brk(2)*.

## SEE ALSO

*brk(2)*, *malloc(3X)*, *stdio(3S)*.

## CHANGE HISTORY

## Issue 1

This function is not included in the SVID. It comes from UNIX System V Release 2.0.





## NAME

erf, erfc — error function and complementary error functions (OPTIONAL)

## SYNOPSIS

```
#include <math.h>
```

```
double erf (x)
```

```
double x;
```

```
double erfc (x)
```

```
double x;
```

## DESCRIPTION

*Erf* returns the error function of  $x$ , defined as  $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ .

*Erfc* returns  $1.0 - \text{erf}(x)$ .

## SEE ALSO

exp(3M).

## APPLICATION USAGE

*Erfc* is provided because of the extreme loss of relative accuracy if *erf*( $x$ ) is called for large  $x$  and the result subtracted from 1.0.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID. The *optional* mathematical group is mandatory in SVID.





## NAME

*exp*, *log*, *log10*, *pow*, *sqrt* — exponential, logarithm, power, square root functions  
(OPTIONAL)

## SYNOPSIS

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;
```

## DESCRIPTION

*Exp* returns  $e^x$ .

*Log* returns the natural logarithm of  $x$ . The value of  $x$  must be positive.

*Log10* returns the logarithm base ten of  $x$ . The value of  $x$  must be positive.

*Pow* returns  $x^y$ . If  $x$  is zero (0),  $y$  must be positive. If  $x$  is negative,  $y$  must be an integer value.

*Sqrt* returns the non-negative square root of  $x$ . The value of  $x$  may not be negative.

## RETURN VALUE

*Exp* returns HUGE when the correct value would overflow, or 0 when the correct value would underflow, and sets *errno* to [ERANGE].

*Log* and *log10* return —HUGE and set *errno* to [EDOM] when  $x$  is non-positive. A message indicating DOMAIN error (or SING error when  $x$  is 0) is printed on the standard error output.

*Pow* returns 0 and sets *errno* to [EDOM] when  $x$  is 0 and  $y$  is non-positive, or when  $x$  is negative and  $y$  is not an integer. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow or underflow, *pow* returns  $\pm$ HUGE or 0 respectively, and sets *errno* to [ERANGE].

*Sqrt* returns 0 and sets *errno* to [EDOM] when  $x$  is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

## SEE ALSO

*hypot*(3M), *matherr*(3M), *sinh*(3M).

## FUTURE DIRECTIONS

A macro `HUGE_VAL` will be defined in the header file `<math.h>`. This macro will call a function which will either return  $+\infty$  on a system supporting IEEE P754 standard or `+{MAXDOUBLE}` on a system that does not support the IEEE P754 standard.

If the correct value overflows, *exp* will return `HUGE_VAL`.

*Log* and *log10* will return `—HUGE_VAL` when *x* is not positive.

*Sqrt* will return -0 when the value of *x* is -0.

The return value of *pow* will be negative `HUGE_VAL` when an illegal combination of input arguments is passed to *pow*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID. The *optional* mathematical group is mandatory in SVID.

## Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

In the paragraph of **RETURN VALUES** describing the return value of *log* and *log10*, the return value has been changed from `HUGE` to `—HUGE`.

## NAME

*fclose*, *fflush* — close or flush a stream

## SYNOPSIS

```
#include <stdio.h>
```

```
int fclose (stream)
```

```
FILE *stream;
```

```
int fflush (stream)
```

```
FILE *stream;
```

## DESCRIPTION

*Fclose* flushes the named stream, and the stream is closed. *Fclose* is performed automatically for all open files upon calling *exit*(2).

*Fflush* flushes the named stream, which remains open.

When a stream is flushed, if it is open for writing, any buffered data for the stream is written out. Otherwise, any buffered data is discarded.

## RETURN VALUE

These functions return 0 for success, and EOF if any error (such as trying to write to a file that has not been opened for writing) was detected.

## SEE ALSO

*close*(2), *exit*(2), *fopen*(3S), *setbuf*(3C).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

The **DESCRIPTION** has been rewritten to improve clarity.

The reference to *fflush* flushing a named stream has been added to reflect the behaviour of System V Release 2.0.





## NAME

error, feof, clearerr, fileno — stream status inquiries

## SYNOPSIS

```
#include <stdio.h>

int error (stream)
FILE *stream;

int feof (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno (stream)
FILE *stream;
```

## DESCRIPTION

*Error* returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*; otherwise zero (0).

*Feof* returns non-zero when EOF has previously been detected reading the named input *stream*, otherwise zero.

*Clearerr* resets the error indicator and EOF indicator to zero on the named *stream*.

*Fileno* returns the integer file descriptor associated with the named *stream*, see *open*(2).

## SEE ALSO

*open*(2), *fopen*(3S).

## APPLICATIONS USAGE

All these functions are implemented as macros; they cannot be declared or redeclared.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

floor, ceil, fmod, fabs — floor, ceiling, remainder, absolute value functions  
(OPTIONAL)

## SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

## DESCRIPTION

*Floor* returns the largest integer (as a double-precision number) not greater than  $x$ .

*Ceil* returns the smallest integer not less than  $x$ .

*Fmod* returns the floating-point remainder of the division of  $x$  by  $y$ : zero if  $y$  is zero or if  $x/y$  would overflow; otherwise the number  $f$  with the same sign as  $x$ , such that  $x = iy + f$  for some integer  $i$ , and  $|f| < |y|$ .

*Fabs* returns the absolute value of  $x$ ,  $|x|$ .

## SEE ALSO

abs(3C).

## FUTURE DIRECTIONS

*Fmod* will return  $x$  if  $y$  is zero or if  $x/y$  would overflow.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID. The *optional* mathematical group is mandatory in SVID.



## NAME

fopen, freopen, fdopen — open a stream

## SYNOPSIS

```
#include <stdio.h>

FILE *fopen (file-name, type)
char *file-name, *type;

FILE *freopen (file-name, type, stream)
char *file-name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

## DESCRIPTION

*Fopen* opens the file named by *file-name* and associates a *stream* with it. *Fopen* returns a pointer to the *FILE* structure associated with the *stream*.

*File-name* points to a character string that contains the name of the file to be opened.

*Type* is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

*Freopen* substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the *FILE* structure associated with *stream*.

*Freopen* is typically used to attach the preopened *streams* associated with *stdin*, *stdout* and *stderr* to other files.

*Fdopen* associates a *stream* with a file descriptor. File descriptors are obtained from *open(2)*, *dup(2)*, *creat(2)* or *pipe(2)*, which open files but do not return pointers to a *FILE* structure *stream*. Streams are necessary input for many of the Section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when *type* is *a* or *a+*), it is impossible to overwrite information already in the file. *Fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the



file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file.

### RETURN VALUE

*Fopen* and *freopen* return a NULL pointer if *file-name* cannot be accessed and the external variable *errno* may contain any of the values listed for *open(2)*.

### SEE ALSO

*creat(2)*, *dup(2)*, *open(2)*, *pipe(2)*, *fclose(3S)*, *fseek(3S)*.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID.

## NAME

fread, fwrite — binary input/output

## SYNOPSIS

```
#include <stdio.h>

#include <sys/types.h>

int fread (ptr, size, nitems, stream)
char *ptr;
size_t size;
int nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
size_t size;
int nitems;
FILE *stream;
```

## DESCRIPTION

*Fread* copies, into an array pointed to by *ptr*, *nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *Fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

*Fwrite* appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. *Fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *Fwrite* does not change the contents of the array pointed to by *ptr*. *Fwrite* increments the file pointer in *stream*, if defined, by the number of bytes written.

## RETURN VALUE

*Fread* and *fwrite* return the number of items read or written. If *size* or *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

## SEE ALSO

read(2), write(2), ferror(3S), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S).

## APPLICATION USAGE

The argument *size* is typically *sizeof(\*ptr)* where the C operator *sizeof* gives the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

*Ferror(3S)* or *feof(3S)* must be used to distinguish between an error condition and an end-of-file condition.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID with the following changes:

The parameters of *fread* and *fwrite* that are declared to be of type *size\_t* are of type *int* in the SVID. Also, to define this type the header file `<sys/types.h>` is specified for inclusion. This is a FUTURE DIRECTION in the SVID.



## NAME

frexp, ldexp, modf — manipulate parts of floating-point numbers

## SYNOPSIS

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;
```

## DESCRIPTION

Every non-zero number can be written uniquely as  $x \cdot 2^n$ , where the "mantissa" (fraction)  $x$  is in the range  $0.5 = |x| < 1.0$ , and the "exponent"  $n$  is an integer. *Frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero (0), both results returned by *frexp* are zero.

*Ldexp* returns the quantity  $value \cdot 2^{exp}$ .

*Modf* returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

## RETURN VALUE

If *ldexp* would cause overflow,  $\pm$ HUGE is returned (according to the sign of *value*), and *errno* is set to [ERANGE].

If *ldexp* would cause underflow, 0 is returned and *errno* is set to [ERANGE].

## FUTURE DIRECTIONS

A macro HUGE\_VAL will be defined in the header file `<math.h>`. This macro will call a function which will either return  $-\infty$  on a system supporting the IEEE P754 standard, or  $+\{\text{MAXDOUBLE}\}$  on a system which does not.

The return value *ldexp* will be  $\pm$ HUGE\_VAL (according to the sign of *value*) in case of overflow.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.



## NAME

*fseek*, *rewind*, *ftell* — reposition a file pointer in a stream

## SYNOPSIS

```
#include <unistd.h>
#include <stdio.h>
```

```
int fseek (stream, offset, ptrname)
```

```
FILE *stream;
```

```
long offset;
```

```
int ptrname;
```

```
void rewind (stream)
```

```
FILE *stream;
```

```
long ftell (stream)
```

```
FILE *stream;
```

## DESCRIPTION

*Fseek* sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, respectively, as *ptrname* takes the value *SEEK\_SET*, *SEEK\_CUR* or *SEEK\_END*.

*Rewind(stream)* is equivalent to *fseek(stream, 0L, 0)*, except that no value is returned.

*Fseek* and *rewind* undo any effects of *ungetc(3S)*.

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

*Ftell* returns the offset of the current byte relative to the beginning of the file associated with the named *stream*. The offset is always measured in bytes.

## SEE ALSO

*lseek(2)*, *fopen(3S)*, *ungetc(3S)*.

## RETURN VALUE

*Fseek* returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; particularly, *fseek* may not be used on a terminal, or a file opened via *popen(3S)*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The SVID does not use the symbolic values *SEEK\_SET*, *SEEK\_CUR* and *SEEK\_END* for *ptrname*.





## NAME

*ftw* — walk a file tree

## SYNOPSIS

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ();
int depth;
```

## DESCRIPTION

*Ftw* recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a *stat* structure (see *stat(2)*) containing information about the object, and an integer. Possible values of the integer, defined in the *<ftw.h>* header file, are *FTW\_F* for a file, *FTW\_D* for a directory, *FTW\_DNR* for a directory that cannot be read, and *FTW\_NS* for an object for which *stat* could not successfully be executed. If the integer is *FTW\_DNR*, descendants of that directory will not be processed. If the integer is *FTW\_NS*, the *stat* structure will contain garbage. An example of an object that would cause *FTW\_NS* to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

*Ftw* visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a non-zero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero (0). If *fn* returns a non-zero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns -1, and sets the error type in *errno*.

*Ftw* uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. The argument *depth* must be in the range 1 to *{OPEN\_MAX}*. *Ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

## SEE ALSO

*stat(2)*, *malloc(3X)*.

## APPLICATION USAGE

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

*Ftw* uses *malloc(3X)* to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a non-zero value at its next invocation.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

In the last paragraph of the **DESCRIPTION**, the description of the range of *depth* has been clarified.



## NAME

gamma, signgam — log gamma function (OPTIONAL)

## SYNOPSIS

```
#include <math.h>

double gamma (x)
double x;

extern int signgam;
```

## DESCRIPTION

*Gamma* returns  $\ln(|\Gamma(x)|)$ , where  $\Gamma(x)$  is defined as  $\int_0^{\infty} e^{-t} t^{x-1} dt$ . The sign of  $\Gamma(x)$  is returned in the external integer *signgam*. The argument *x* may not be a non-positive integer.

The following C program fragment might be used to calculate  $\Gamma$ :

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
    error();
y = signgam * exp(y);
```

where LN\_MAXDOUBLE is the least value that causes *exp*(3M) to return a range error, and is defined in the *<values.h>* header file.

## RETURN VALUE

For non-positive integer arguments HUGE is returned, and *errno* is set to [EDOM]. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns HUGE and sets *errno* to [ERANGE].

These error-handling procedures may be changed with the function *matherr*(3M).

## SEE ALSO

*exp*(3M), *matherr*(3M).

## FUTURE DIRECTIONS

A macro HUGE\_VAL will be defined in the header file *<math.h>*. This macro will call a function which will either return  $+\infty$  on a system supporting IEEE P754 standard or +(MAXDOUBLE) on a system that does not support the IEEE P754 standard.

If the correct value overflows, *gamma* will return HUGE\_VAL.

## CHANGE HISTORY

## Issue 1

The *optional* mathematical group is mandatory in SVID.

Derived from the entry in Issue 1 of the SVID, with the following change:

*Signgam* has been added to the **NAME** section.

## Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

The sign of the integer arguments in the **RETURN VALUE** section have been changed from non-negative to non-positive.



## NAME

getc, getchar, fgetc, getw — get character or word from a stream

## SYNOPSIS

```
#include <stdio.h>
```

```
int getc (stream)
```

```
FILE *stream;
```

```
int getchar ()
```

```
int fgetc (stream)
```

```
FILE *stream;
```

```
int getw (stream)
```

```
FILE *stream;
```

## DESCRIPTION

*Getc* returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *Getchar* is defined as *getc(stdin)*. *Getc* and *getchar* are macros.

*Fgetc* behaves like *getc*, but is a function rather than a macro. *Fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

*Getw* returns the next word (i.e. integer) from the named input *stream*. *Getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. *Getw* assumes no special alignment in the file.

## RETURN VALUE

These functions return the constant EOF at end-of-file or upon an error. Because EOF is a valid integer, *error(3S)* should be used to detect *getw* errors.

## SEE ALSO

*fclose(3S)*, *ferror(3S)*, *fopen(3S)*, *fread(3S)*, *gets(3S)*, *putc(3S)*, *scanf(3S)*.

## APPLICATION USAGE

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, *getc(\*f+ +)* does not work sensibly. *Fgetc* should be used instead.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

getcwd — get path-name of current working directory

## SYNOPSIS

```
char *getcwd (buf, size)
char *buf;
int size;
```

## DESCRIPTION

*Getcwd* returns a pointer to the current directory path-name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc(3X)*. In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*.

## ERRORS

[EINVAL]	<i>size</i> is zero
[ENOMEM]	no space available
[ERANGE]	<i>size</i> not large enough to hold the path-name.

## RETURN VALUE

Returns NULL with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

## SEE ALSO

*malloc(3X)*.

## APPLICATION USAGE

Invoking *getcwd* with *buf* as a null pointer is not recommended as this functionality is not included in Issue 2 of the SVID and may be subject to later withdrawal.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

The **APPLICATION NOTE** relating the use of *buf* as a null pointer was added.





NAME

getenv — return value for environment name

SYNOPSIS

```
char *getenv (name)
char *name;
```

DESCRIPTION

*Getenv* searches the environment list for a string of the form "*name=value*", and returns a pointer to the *value* in the current environment if such a string is present. Otherwise a NULL pointer is returned.

SEE ALSO

exec(2), putenv(3C).

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID.

Issue 2

Spaces incorrectly surrounding the "=" sign have been removed.



## NAME

endgrent, fgetgrent, getgrent, getgrgid, getgrnam, setgrent — get group file entry

## SYNOPSIS

```
#include <grp.h>
#include <stdio.h>

struct group *getgrent ()

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ()

void endgrent ()

struct group *fgetgrent (f)
FILE *f;
```

## DESCRIPTION

*Getgrent*, *getgrgid* and *getgrnam* each return pointers to a *struct group* with the following structure containing the broken-out fields of a line in the group file.

```
struct group
    char      *gr_name;
    char      *gr_passwd;
    int       gr_gid;
    char      **gr_mem;
```

The members of this structure are:

<i>gr_name</i>	The name of the group.
<i>gr_passwd</i>	The encrypted password of the group.
<i>gr_gid</i>	The numerical group ID.
<i>gr_mem</i>	Null-terminated vector of pointers to the individual member names.

*Getgrent* reads the next line of the file, so successive calls may be used to search the entire file. *Getgrgid* and *getgrnam* search from the beginning of the file until a matching *gid* or *name* is found or EOF is encountered.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *Endgrent* may be called to close the group file when processing is complete.

The routine *fgetgrent* returns a pointer to the next *group* structure in the file *f*; this file must have the format of */etc/group*.

## FILES

*/etc/group*

## SEE ALSO

getlogin(3C), getpwent(3C), group(4), grp(5).

## RETURN VALUE

A NULL pointer is returned on EOF or error.



## APPLICATION USAGE

All information is contained in a static area so it must be copied if it is to be saved.

There is no convenient way to enter a password into */etc/group*. Use of group passwords is not encouraged, because by their very nature they encourage poor security practices. Group passwords may disappear in future.

## CHANGE HISTORY

### Issue 1

This function is not in Issue 1 of the SVID. It comes from System V Release 2.0.

### Issue 2

Aligned with the entry in the SD\_LIB section of Issue 2 of the SVID by adding the *fgetgrent* function.

## NAME

getlogin — get login name

## SYNOPSIS

char \*getlogin ()

## DESCRIPTION

*Getlogin* returns a pointer to the login name as found in */etc/utmp*. It may be used in conjunction with *getpwnam*(3C) to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns NULL. The correct procedure for determining the login name is to call *cuserid*(3S), or to call *getlogin* and if it fails, to call *getpwuid*(3C).

## FILES

*/etc/utmp*

## SEE ALSO

*cuserid*(3S), *getgrent*(3C), *getpwent*(3C), *getpwnam*(3C), *getpwuid*(3C), *utmp*(5).

## RETURN VALUE

Returns NULL if name not found.

## APPLICATION USAGE

The return values point to static data whose content is overwritten by each call.

## CHANGE HISTORY

## Issue 1

This function is not included in Issue 1 of the SVID. It comes from System V Release 2.0.

## Issue 2

Aligned with the entry in the SD\_LIB section of Issue 2 of the SVID.





## NAME

`getopt` — get option letter from argument vector

## SYNOPSIS

```
int getopt (argc, argv, optstring)
int argc;
char *argv [], *optstring;

extern char *optarg;
extern int optind, opterr;
```

## DESCRIPTION

*Getopt* is a command line parser. It returns the next option letter in *argv* that matches a letter in *optstring*. *Optstring* is a string of recognised option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from *getopt*.

*Getopt* places in *optind* the *argv* index of the next argument to be processed. The external variable *optind* is initialised to 1 before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns EOF. The special option `--` may be used to delimit the end of the options; EOF will be returned, and `--` will be skipped.

## RETURN VALUE

*Getopt* prints an error message on *stderr* and returns a question mark (?) when it encounters an option letter not included in *optstring*. This error message may be disabled by setting *opterr* to zero (0).

## EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
    int c;
    int bflg, aflg, errflg;
    char *ifile;
    char *ofile;
    extern char *optarg;
    extern int optind;
    .
    .
    while ((c = getopt(argc, argv, "abf:o:")) != EOF)
        switch (c) {
            case 'a':
                if (bflg)
                    errflg++;
                else
                    aflg++;
                break;
```

```
        case 'b':
            if (aflag)
                errflg++;
            else
                bproc( );
            break;
        case 'f':
            ifile = optarg;
            break;
        case 'o':
            ofile = optarg;
            break;
        case '?:
            errflg++;
    }
    if (errflg) {
        fprintf(stderr, "usage: . . . ");
        exit (2);
    }
    for ( ; optind < argc; optind++) {
        if (access(argv[optind], 4)) {
            .
            .
            .
        }
    }
```

#### FUTURE DIRECTIONS

*Getopt* will be enhanced to enforce all rules of the System V Command Syntax Standard.

#### CHANGE HISTORY

##### Issue 1

Derived from the entry in Issue 1 of the SVID.

## NAME

getpass — read a password

## SYNOPSIS

```
char *getpass (prompt)
char *prompt;
```

## DESCRIPTION

The *getpass* routine opens a terminal device using file */dev/tty*, prompts to that device with the null-terminated string *prompt*, disables echoing, reads a string of characters up to the next newline character or EOF, re-enables echoing and closes the device file.

## FILES

*/dev/tty*

## RETURN VALUE

A pointer is returned to a null-terminated string of at most {PASS\_MAX} characters that were read from the terminal device. This string entry is not returned in encoded form. If an error is encountered in this process, a NULL pointer is returned.

## SEE ALSO

crypt(3C).

## APPLICATION USAGE

The return value points to static data whose content is overwritten by each call.

## CHANGE HISTORY

## Issue 1

This function is not included in Issue 1 of the SVID. It comes from System V Release 2.0.

## Issue 2

Aligned with the entry in the SD\_LIB section of Issue 2 of the SVID.





## NAME

getpw — get name from UID

## SYNOPSIS

```
getpw (uid, buf)
int uid;
char *buf;
```

## DESCRIPTION

*Getpw* searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

This routine is included only for compatibility with prior systems and should not be used, see *getpwent*(3C) for routines to use instead.

## FILES

/etc/passwd

## SEE ALSO

getpwent(3C), passwd(5).

## RETURN VALUE

Non-zero return on error.

## CHANGE HISTORY

## Issue 1

This function is not included in Issue 1 of the SVID. It comes from System V Release 2.0.





## NAME

endpwent, fgetpwent, getpwent, getpwuid, getpwnam, setpwent — get password file entry

## SYNOPSIS

```
#include <pwd.h>
#include <stdio.h>

struct passwd *getpwent ()

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ()

void endpwent ()

struct passwd *fgetpwent (f)
FILE *f;
```

## DESCRIPTION

*Getpwent*, *getpwuid* and *getpwnam* each return a pointer to a *struct passwd* containing the broken-out fields of a line in the password file:

```
struct passwd
    char *pw_name;
    char *pw_passwd;
    int pw_uid;
    int pw_gid;
    char *pw_age;
    char *pw_comment;
    char *pw_gecos;
    char *pw_dir;
    char *pw_shell;

struct comment
    char *c_dept;
    char *c_name;
    char *c_acct;
    char *c_bin;
```

The *pw\_comment* field is unused; the others have meanings described in *pwd(5)*.

*Getpwent* reads the next line in the file, so successive calls can be used to search the entire file. *Getpwuid* and *getpwnam* search from the beginning of the file until a matching *uid* or *name* is found, or EOF is encountered.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *Endpwent* may be called to close the password file when processing is complete.

The routine *fgetpwent* returns a pointer to the next *passwd* structure in file *f*, which must have the format of */etc/passwd*.

### FILES

/etc/passwd

### SEE ALSO

getlogin(3C), getgrent(3C), passwd(4), pwd(5).

### RETURN VALUE

NULL pointer returned on EOF or error.

### APPLICATION USAGE

All information is contained in a static area so it must be copied if it is to be saved.

The use of *pw\_gecos* is discouraged as its contents are not clearly defined.

The use of *struct comment* is discouraged as its functionality is not described. It may be subject to later withdrawal.

### CHANGE HISTORY

#### Issue 1

This function is not included in Issue 1 of the SVID. It comes from System V Release 2.0.

#### Issue 2

Aligned with the entry in the SD\_LIB section of Issue 2 of the SVID by adding the *fgetpwent* function.

## NAME

gets, fgets — get a string from a stream

## SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

## DESCRIPTION

*Gets* reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a NULL character.

*Fgets* reads characters from the *stream* into the array pointed to by *s*, until *n*—1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a NULL character.

## RETURN VALUE

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

## SEE ALSO

ferror(3S), fopen(3S), fread(3S),getc(3S), scanf(3S).

## APPLICATION USAGE

Reading a line that is too long through *gets* causes *gets* to break. The use of *fgets* is recommended.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname — access utmp file entry

## SYNOPSIS

```
#include <sys/types.h>
#include <utmp.h>

struct utmp *getutent ()

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ()

void endutent ()

void utmpname (file)
char *file;
```

## DESCRIPTION

*Getutent*, *getutid* and *getutline* each return a pointer to a structure containing the following members:

*struct utmp*

```
char          ut_user[8];      /* User login name */
char          ut_id[4];        /* /etc/inittab id (usually line #) */
char          ut_line[12];     /* device name (console, lnxx) */
short         ut_pid;          /* process ID */
short         ut_type;         /* type of entry */
struct exit_status ut_exit;    /* The exit status of a process
                               * marked as DEAD_PROCESS. */
time_t        ut_time;        /* time entry made */
```

*struct exit\_status* contains

```
short e_termination; /* Process termination status */
short e_exit;         /* Process exit status */
```

*Getutent* reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

*Getutid* searches forward from the current point in the *utmp* file until it finds an entry with a *ut\_type* matching *id->ut\_type* if the type specified is *RUN\_LVL*, *BOOT\_TIME*, *OLD\_TIME* or *NEW\_TIME*. If the type specified in *ut\_type* is *INIT\_PROCESS*, *LOGIN\_PROCESS*, *USER\_PROCESS* or *DEAD\_PROCESS*, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut\_id* field matches *id->ut\_id*. If the end of file is reached without a match, it fails.

*Getutline* searches forward from the current point in the *utmp* file until it finds an entry of the type *LOGIN\_PROCESS* or *USER\_PROCESS* which also has a *ut\_line* string matching the *line->ut\_line* string. If the end of file is reached without a

match, it fails.

*Pututline* writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

*Setutent* resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

*Endutent* closes the currently open file.

*Utmpname* allows the user to change the name of the file examined, from */etc/utmp* to any other file. It is most often expected that this other file will be */etc/wtmp*. If the file doesn't exist, this will not be apparent until the first attempt to reference the file is made. *Utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

#### FILES

*/etc/utmp*  
*/etc/wtmp*

#### SEE ALSO

*ttyslot(3C)*, *utmp(5)*.

#### RETURN VALUE

A NULL pointer is returned upon failure to read, whether because of permissions or having reached the end of file, or upon failure to write.

#### APPLICATION USAGE

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again.

There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* if it finds that it isn't already at the correct place in the file will not alter the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

The use of *ut\_time* and *ut\_exit* are discouraged as they are not in the SVID and may be subject to later withdrawal.

The constants used as array sizes in *struct utmp* represent the values known on all current X/OPEN implementations. As these values are system-dependent, for additional portability, applications may make use of *sizeof* to determine the actual array sizes.



CHANGE HISTORY

Issue 1

This function is not included in Issue 1 of the SVID. It is derived from UNIX System V Release 2.0.

Issue 2

Aligned with the entry in the entry in the SD\_LIB section of Issue 2 of the SVID by applying the following changes:

The call for inclusion of `<sys/types.h>` has been added.

The paragraph in **APPLICATION USAGE** relating to *ut\_time* and *ut\_exit* has been added.

The paragraph in **APPLICATION USAGE** relating to array sizes in *struct utmp* has been added.



## NAME

hsearch, hcreate, hdestroy — manage hash search tables

## SYNOPSIS

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ()
```

## DESCRIPTION

*Hsearch* is a hash-table search routine generalised from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *Item* is a structure of type ENTRY (defined in the <search.h> header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *Action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. ENTER indicates that the item should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

*Hcreate* allocates sufficient space for the table, and must be called before *hsearch* is used. *Nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

*Hdestroy* destroys the search table, and may be followed by another call to *hcreate*.

## RETURN VALUE

*Hsearch* returns a NULL pointer if either the action is FIND and the item could not be found or the action is ENTER and the table is full.

*Hcreate* returns zero (0) if it cannot allocate sufficient space for the table.

## SEE ALSO

bsearch(3C), lsearch(3C), malloc(3X), string(3C), tsearch(3C), search(5).

## APPLICATION USAGE

*Hsearch* and *hcreate* use *malloc(3X)* to allocate space.

## EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

```
#include <stdio.h>
#include <search.h>

struct info {          /* this is the info stored in the table */
    int age, room;      /* other than the key. */
};
#define NUM_EMPL      5000    /* # of elements in search table */
```



```

main( )
{
    /* space to store strings */
    char string_space[NUM_EMPL*20];
    /* space to store employee info */
    struct info info_space[NUM_EMPL];
    /* next avail space in string_space */
    char *str_ptr = string_space;
    /* next avail space in info_space */
    struct info *info_ptr = info_space;
    ENTRY item, *found_item, *hsearch( );
    /* name to look for in table */
    char name_to_find[30];
    int i = 0;

    /* create table */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
        &info_ptr->room) != EOF && i++ < NUM_EMPL) {
        /* put info in structure, and structure in item */
        item.key = str_ptr;
        item.data = (char *)info_ptr;
        str_ptr += strlen(str_ptr) + 1;
        info_ptr++;
        /* put item into table */
        (void) hsearch(item, ENTER);
    }

    /* access table */
    item.key = name_to_find;
    while (scanf("%s", item.key) != EOF) {
        if ((found_item = hsearch(item, FIND)) != NULL) {
            /* if item is in the table */
            (void)printf("found %s, age = %d, room = %d\n",
                found_item->key,
                ((struct info *)found_item->data)->age,
                ((struct info *)found_item->data)->room);
        } else {
            (void)printf("no such employee %s\n",
                name_to_find);
        }
    }
}

```

**FUTURE DIRECTIONS**

The restriction of having only one hash search table active at any given time will be removed.

**CHANGE HISTORY****Issue 1**

Derived from the entry in Issue 1 of the SVID.

**Issue 2**

Aligned with the entry in Issue 2 of the SVID by applying the following change:  
References to the source of *hsearch* have been deleted.





## NAME

hypot — Euclidean distance function (OPTIONAL)

## SYNOPSIS

```
#include <math.h>
```

```
double hypot (x, y)  
double x, y;
```

## DESCRIPTION

*Hypot* returns

$$\sqrt{x * x + y * y},$$

taking precautions against unwarranted overflows.

## RETURN VALUE

When the correct value would overflow, *hypot* returns HUGE and sets *errno* to [ERANGE].

These error-handling procedures may be changed with the function *matherr*(3M).

## SEE ALSO

*matherr*(3M).

## FUTURE DIRECTIONS

A macro HUGE\_VAL will be defined in the header file <math.h>. This macro will call a function which will either return  $+\infty$  on a system supporting IEEE P754 standard or +{MAXDOUBLE} on a system that does not support the IEEE P754 standard.

If the correct value overflows, *hypot* will return HUGE\_VAL.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID. The *optional* mathematical group is mandatory in SVID.



## NAME

*l3tol*, *l3tol3* — convert between 3-byte integers and long integers

## SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void l3tol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

## DESCRIPTION

*l3tol* converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

*l3tol3* performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

## APPLICATION USAGE

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

## CHANGE HISTORY

## Issue 1

This function is not included in the SVID. It comes from UNIX System V Release 2.0.





## NAME

lockf — record locking on files

## SYNOPSIS

```
#include <unistd.h>

int lockf (fildes, function, size)
long size;
int fildes, function;
```

## DESCRIPTION

The *lockf* routine will allow sections of a file to be locked. *Lockf* calls from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. See *fcntl(2)* for more information about record locking.

*Fildes* is an open file descriptor. The file descriptor must have *O\_WRONLY* or *O\_RDWR* permission in order to establish a lock with this function call.

*Function* is a control value which specifies the action to be taken. The permissible values for *function* are defined in *<unistd.h>* as follows:

<i>F_ULOCK</i>	0	/* Unlock a previously locked section */
<i>F_LOCK</i>	1	/* Lock a section for exclusive use */
<i>F_TLOCK</i>	2	/* Test and lock a section for exclusive use */
<i>F_TEST</i>	3	/* Test section for other processes' locks */

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

*F\_TEST* is used to detect if a lock by another process is present on the specified section. *F\_LOCK* and *F\_TLOCK* both lock a section of a file if the section is available. *F\_ULOCK* removes locks from a section of the file.

*Size* is the number of contiguous bytes to be locked or unlocked. The resource to be locked or unlocked starts at the current offset in the file, and extends forward, for a positive *size*, or backwards for a negative *size* (the preceding byte, up to but not including the current offset). If *size* is zero (0) the section from the current offset through *{FCHR\_MAX}* is locked (i.e., from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked, as locks may exist past the end-of-file.

The sections locked with *F\_LOCK* or *F\_TLOCK* may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections would occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

*F\_LOCK* and *F\_TLOCK* requests differ only by the action taken if the resource is not available. *F\_LOCK* will cause the calling process to sleep until the resource is available. *F\_TLOCK* will cause the function to return a *-1* and set *errno* to *[EACCES]* or *[EAGAIN]* if the section is already locked by another process.

F\_ULOCK requests may release, in whole or in part, one or more locked sections controlled by the process. Locked sections will be unlocked starting at the point of the file offset through *size* bytes, or to the end of file, if *size* is 0. When sections are not fully released, the remaining sections are still locked by the process. For example, releasing the center portion of a locked section will leave the portions of the section before and after it locked, and requires an additional element in the table of active locks. If this table is full, -1 is returned, *errno* is set to [EDEADLK] and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to *lockf*(3C), or *fcntl*(2) scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) routine may be used to provide a timeout facility in applications which require this facility.

## ERRORS

The *lockf* routine will fail if one or more of the following are true:

- |           |   |
|-----------|---|
| [EBADF]   | <i>filides</i> is not a valid file descriptor.  |
| [EACCES]  |   |
| [EGAIN]   | due to implementation details, one of these errors will be returned for the following condition: <i>function</i> is F_TLOCK or F_TEST and the section is already locked by another process. |
| [EDEADLK] | <i>function</i> is F_LOCK and a deadlock would occur. Also the <i>function</i> is F_LOCK, F_TLOCK or F_ULOCK and the number of entries in the system lock table would exceed {LOCK_MAX}.    |

## RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## SEE ALSO

*chmod*(2), *close*(2), *creat*(2), *fcntl*(2), *open*(2), *read*(2), *write*(2), *unistd*(5).

## APPLICATION USAGE

Record and file locking should not be used in combination with the *stdio*(3S) routines: *fopen*(3S), *fread*(3S), *fwrite*(3S), etc. Instead, the more primitive, non-buffered routines, e.g., *open*(2) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O package is the most common source of unexpected buffering.

## CHANGE HISTORY

### Issue 1

Derived from the entry in Issue 1 of the SVID.



**Issue 2**

Except that the **FUTURE DIRECTION** on mandatory or forced-mode file locking has been dropped, aligned with Issue 2 of the SVID by applying the following changes:

The reference to [EACCES] has been included.

The reference to F\_TLOCK in the [EDEADLK] sub-section of the **ERRORS** section has been corrected.



NAME

logname — return login name of user

SYNOPSIS

char \*logname ()

DESCRIPTION

*Logname* returns a pointer to the null-terminated login name; it extracts the LOGNAME variable from the user's environment.

SEE ALSO

environ(5).

APPLICATION USAGE

The return values point to static data whose content is overwritten by each call.

CHANGE HISTORY

Issue 1

This function is not in Issue 1 of the SVID. It is derived from UNIX System V Release 2.0.

Issue 2

The FILES section was deleted.





## NAME

*lsearch*, *lfind* — linear search and update

## SYNOPSIS

```
#include <search.h>

char *lsearch (key, base, nelp, width, compar)
char *key;
char *base;
unsigned *nelp;
unsigned width;
int (*compar)();

char *lfind (key, base, nelp, width, compar)
char *key;
char *base;
unsigned *nelp;
unsigned width;
int (*compar)();
```

## DESCRIPTION

*lsearch* is a linear search routine generalised from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. *Key* points to the datum to be sought in the table. *Base* points to the first element in the table. *Width* is the size of an element in bytes. *Nelp* points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. *Compar* is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero (0) if the elements are equal and non-zero otherwise.

*lfind* is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

## RETURN VALUE

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

## SEE ALSO

*bsearch*(3C), *hsearch*(3C), *tsearch*(3C), *search*(5).

## APPLICATION USAGE

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Undefined results can occur if there is not enough room in the table to add a new item.

### EXAMPLE

This fragment will read in  $\leq$  TABSIZE strings of length  $\leq$  ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>
```

```
#define TABSIZE 50
#define ELSIZE 120
```

```
char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
unsigned nel = 0;
int strcmp( );

...
while (fgets(line, ELSIZE, stdin) != NULL &&
      nel < TABSIZE)
    (void) lsearch(line, (char *)tab, &nel,
                  ELSIZE, strcmp);

...
```

### FUTURE DIRECTIONS

A NULL pointer will be returned with *errno* set appropriately, if there is not enough room in the table to add a new item.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID.

#### Issue 2

The type of the pointer of *width* in the SYNOPSIS section has been corrected.



## NAME

malloc, free, realloc, calloc, mallopt, mallinfo — fast main memory allocator

## SYNOPSIS

```
#include <malloc.h>

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo ()
```

## DESCRIPTION

*Malloc* and *free* provide a simple general-purpose memory allocation package.

*Malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

*Realloc* changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

*Calloc* allocates space for an array of *nelem* elements of size *elsize*. The space is initialised to zeros.

*Mallopt* provides for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST	Set <i>maxfast</i> to <i>value</i> . The algorithm allocates all blocks below the size of <i>maxfast</i> in large groups and then does them out very quickly. The default value for <i>maxfast</i> is zero (0).
M_NLBLKS	Set <i>numlblks</i> to <i>value</i> . The above mentioned "large groups" each contain <i>numlblks</i> blocks. <i>Numlblks</i> must be greater than one. The default value for <i>numlblks</i> is 100.
M_GRAIN	Set <i>grain</i> to <i>value</i> . The sizes of all blocks smaller than <i>maxfast</i> are considered to be rounded up to the nearest multiple of <i>grain</i> . <i>Grain</i> must be greater than zero. The default value of <i>grain</i> is the smallest number of bytes which will allow alignment of any data type. <i>Value</i> will be rounded up to a multiple of the default when <i>grain</i> is set.

**M\_KEEP** Preserve data in a freed block until the next *malloc*, *realloc*, or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended.

These values are defined in the `<malloc.h>` header file.

*Mallopt* may be called repeatedly, but may not be called after the first small block is allocated.

*Mallinfo* provides information describing space usage. It returns the structure *mallinfo* which includes the following members:

int arena;	/* total space in arena */
int ordblks;	/* number of ordinary blocks */
int smblks;	/* number of small blocks */
int hblkh;	/* space in holding block headers */
int hblks;	/* number of holding blocks */
int usmblks;	/* space in small blocks in use */
int fsmblks;	/* space in free small blocks */
int uordblks;	/* space in ordinary blocks in use */
int fordblks;	/* space in free ordinary blocks */
int keepcost;	/* space penalty if keep option */
	/* is used */

This structure is defined in the `<malloc.h>` header file.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

### RETURN VALUE

*Malloc*, *realloc* and *calloc* return a NULL pointer if there is not enough available memory. When *realloc* returns NULL, the block pointed to by *ptr* is left intact. If *mallopt* is called after any allocation or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

### APPLICATION USAGE

This is the SVID version of *malloc*. An older (smaller) form may also exist, providing only the functionality of *malloc*, *free*, *realloc* and *calloc*. If both forms are present, it is the responsibility of the application developers to ensure that the appropriate version is linked into their applications. The system documentation will indicate in which library which version of *malloc* can be found.

*Mallinfo* should only be invoked once storage has been successfully allocated.

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID.

#### Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

In the **DESCRIPTION** of **M\_NLBLKS**, the word "zero" has been changed to "one".



## NAME

`matherr` — error-handling function (OPTIONAL)

## SYNOPSIS

```
#include <math.h>
```

```
int matherr(x)
struct exception *x;
```

## DESCRIPTION

*Matherr* is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors, by including a function named *matherr* in their programs. *Matherr* must be of the form described above. When an error occurs, a pointer to the *exception* structure *x* will be passed to the user-supplied *matherr* function. This structure, which is defined in the `<math.h>` header file includes the following members:

```
int type;
char *name;
double arg1, arg2, retval;
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain error
SING	argument singularity
OVERFLOW	overflow range error
UNDERFLOW	underflow range error
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element *name* points to a string containing the name of the function that incurred the error. The variables *arg1* and *arg2* are the arguments with which the function was invoked. *Retval* is set to the default value that will be returned by the function unless the user's *matherr* sets it to a different value.

If the user's *matherr* function returns non-zero, no error message will be printed, and *errno* will not be set.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the mathematical functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to [EDOM] or [ERANGE] and the program continues.

## FUTURE DIRECTIONS

The mathematical functions which return HUGE or  $\pm$ HUGE on overflow will return HUGE\_VAL or  $\pm$ HUGE\_VAL respectively.

## EXAMPLE

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
    switch (x->type) {
        case DOMAIN:
            /* change sqrt to return sqrt(-arg1), not 0 */
```



```
        if (!strcmp(x->name, "sqrt")) {
            x->retval = sqrt(-x->arg1);
            return (0); /* print message and set errno */
        }
    case SING:
        /* all other domain or sing errors, print message and abort */
        fprintf(stderr, "domain error in %s\n", x->name);
        abort();
    case PLOSS:
        /* print detailed error message */
        fprintf(stderr, "loss of significance in %s(%g) = %g\n",
            x->name, x->arg1, x->retval);
        return (1); /* take no other action */
    }
    return (0); /* all other errors, execute default procedure */
}
```

DEFAULT ERROR HANDLING PROCEDURES						
	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL: y0, y1, yn (arg = 0)	- M, —H	- -	- -	- -	M, 0 -	* -
EXP:	-	-	H	0	-	-
LOG, LOG10: (arg < 0) (arg = 0)	M, —H -	- M, —H	- -	- -	- -	- -
POW: neg ** non-int 0 ** non-pos	- M, 0	- -	±H -	0 -	- -	- -
SQRT:	M, 0	-	-	-	-	-
GAMMA:	-	M, H	H	-	-	-
HYPOT:	-	-	H	-	-	-
SINH:	-	-	±H	-	-	-
COSH:	-	-	H	-	-	-
SIN, COS, TAN: —	-	-	-	M, 0	*	-
ASIN, ACOS, ATAN2: M, 0	-	-	-	-	-	-

ABBREVIATIONS	
*	As much as possible of the value is returned.
M	Message is printed (EDOM error).
H	HUGE is returned.
-H	-HUGE is returned.
±H	HUGE or -HUGE is returned.
0	0 is returned.

CHANGE HISTORY  
Issue 1

Derived from the entry in Issue 1 of the SVID. The *optional* mathematical group is mandatory in the SVID.

## NAME

memccpy, memchr, memcmp, memcpy, memset — memory operations

## SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

## DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a NUL character). They do not check for the overflow of any receiving memory area.

*Memccpy* copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* characters of *s2*.

*Memchr* returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*, or a NULL pointer if *c* does not occur.

*Memcmp* compares its arguments, looking at the first *n* characters only, and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*.

*Memcpy* copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

*Memset* sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

## SEE ALSO

string(3C), memory(5).

## APPLICATION USAGE

All these functions are declared in the `<memory.h>` header file.

*Memcmp* uses native character comparison. The sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

Character movement is performed differently in different implementations. Thus overlapping moves may be unpredictable.

## **.FUTURE DIRECTIONS**

The declarations in `<memory.h>` will move to `<string.h>`.

## **CHANGE HISTORY**

### **Issue 1**

Derived from the entry in Issue 1 of the SVID.



## NAME

mktemp — make a unique file name

## SYNOPSIS

```
char *mktemp (template)
char *template;
```

## DESCRIPTION

*Mktemp* replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing X's; *mktemp* will replace the X's with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

## APPLICATION USAGE

It is possible to run out of letters in which case a NULL pointer is returned.

## SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

## FUTURE DIRECTIONS

A NULL pointer will be returned if a unique name cannot be created.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.



## NAME

monitor — prepare execution profile (OPTIONAL)

## SYNOPSIS

```
#include <mon.h>
```

```
void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)(), (*highpc)();
WORD *buffer;
int bufsize, nfunc;
```

## DESCRIPTION

*Monitor* is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* WORDs (defined in the <mon.h> header file). *Monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *Lowpc* may not equal zero (0) for this use of *monitor*. At most *nfunc* call counts can be kept.

For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
```

```
...
```

```
monitor (((int (*)())2, etext, buf, bufsize, nfunc);
```

*Etext* lies just above all the program text, see *end*(3C).

## SEE ALSO

*profil*(2), *end*(3C).

## CHANGE HISTORY

## Issue 1

This *optional* function is not included in the SVID. It comes from UNIX System V Release 2.0.





## NAME

perror, errno, sys\_errlist, sys\_nerr — system error messages

## SYNOPSIS

```
void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;
```

## DESCRIPTION

*Perror* produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys\_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *sys\_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID. This is "level 2" in SVID, see SUBJECT TO CHANGE in Chapter 1. In the FUTURE DIRECTIONS section, the SVID states: "New error handling routines will be added to support the System V Error Message Standard as a tool for application developers to use".





## NAME

`popen`, `pclose` — initiate pipe to/from a process

## SYNOPSIS

```
#include <stdio.h>
```

```
FILE *popen (command, type)
```

```
char *command, *type;
```

```
int pclose (stream)
```

```
FILE *stream;
```

## DESCRIPTION

The arguments to *popen* are pointers to null-terminated strings containing, respectively, a command line and an I/O mode, either "r" for reading or "w" for writing. *Popen* creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is "w" by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is "r" by reading from the file *stream*.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command may be used as an input filter, and a type "w" as an output filter.

## SEE ALSO

`pipe(2)`, `wait(2)`, `fclose(3S)`, `fopen(3S)`, `system(3S)`.

## RETURN VALUE

*Popen* returns a NULL pointer if files or processes cannot be created, or if the command cannot be executed.

*Pclose* returns `-1` if *stream* is not associated with a *popen* command.

## APPLICATION USAGE

Only one stream opened by *popen* can be in use at once.

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush*, see *fclose(3S)*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

In the SVID the first sentence of the RETURN VALUE section ends: "...or if the shell cannot be accessed".



## NAME

printf, fprintf, sprintf — print formatted output (NLS)

## SYNOPSIS

```
#include <stdio.h>

int printf (format [ , arg ] ... )
char *format;

int fprintf (stream, format [ , arg ] ... )
FILE *stream;
char *format;

int sprintf (s, format [ , arg ] ... )
char *s, *format;
```

## DESCRIPTION

*Printf* places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places output followed by the NULL character ('\0'), in consecutive bytes starting at *\*s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the '\0' in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the *format*. If the *format* is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character '%'. After the '%', the following appear in sequence:

- Zero or more *flags*, which modify the meaning of the conversion specification.

- An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '-', described below, has been given) to the field width.

- A *precision* that gives the minimum number of digits to appear for the 'd', 'o', 'u', 'x', or 'X' conversions, the number of digits to appear after the decimal point for the 'e' and 'f' conversions, the maximum number of significant digits for the 'g' conversion, or the maximum number of characters to be printed from a string in 's' conversion. The precision takes the form of a period ('.') followed by a decimal digit string; a null digit string is treated as zero.

- An optional *l* (ell) specifying that a following 'd', 'o', 'u', 'x', or 'X' conversion character applies to a long integer *arg*. A 'l' before any other conversion character is ignored.

- A character that indicates the type of conversion to be applied.



A field width or precision may be indicated by an asterisk ('\*') instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

- The result of the conversion will be left-justified within the field.
- + The result of a signed conversion will always begin with a sign ('+' or '-').
- blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and '+' flags both appear, the blank flag will be ignored.
- # This flag specifies that the value is to be converted to an "alternate form". For 'c', 'd', 's' and 'u' conversions, the flag has no effect. For 'o' conversion, it increases the precision to force the first digit of the result to be a zero. For 'x' or 'X' conversion, a non-zero result will have "0x" or "0X" prefixed to it. For 'e', 'E', 'f', 'g', and 'G' conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For 'g' and 'G' conversions, trailing zeroes will *not* be removed from the result as they normally are.

The conversion characters and their meanings are:

- d,o,u,x,X The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation ('x' and 'X'), respectively; the letters "abcdef" are used for 'x' conversion and the letters "ABCDEF" for 'X' conversion.
- The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
- f The float or double *arg* is converted to decimal notation in the style "[.]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.
- e,E The float or double *arg* is converted in the style "[.]d.ddde±dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The 'E' format code will produce a number with "E" instead of "e" introducing the exponent. The exponent always contains at least two digits. However, if the value to be printed is greater than or equal to 1E+100, additional exponent digits will be pointed as necessary.
- g,G The float or double *arg* is printed in style 'f' or 'e' (or in style 'E' in the case of a 'G' format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style 'e' will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
- c The character *arg* is printed.

- s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a NULL character ('\0') is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.
- % Print a "%"; no argument is converted.

If the character after the '%' is not a valid conversion character, the results of the conversion are not predictable.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

#### SEE ALSO

*putc*(3S), *scanf*(3S), *stdio*(3S).

#### FUTURE DIRECTIONS

*Printf* will make available character string representation for  $\infty$  and "Not a Number" (NaN: a symbolic entity encoded in floating point format) to support the IEEE P754 standard.

#### EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday, month, day, hour, min);
```

To print  $\pi$  to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

#### CHANGE HISTORY

##### Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The part of the FUTURE DIRECTIONS section which in the SVID reads as follows has been removed:

"System V currently allows leading zero padding to be specified by prepending a zero to the field width. The documentation of this feature will be removed to describe the preferred usage for the future application development. Ultimately this feature will be unsupported and customers will be warned of using an unsupported feature."

The SVID has in fact already incorporated this FUTURE DIRECTION.





## NAME

putc, putchar, fputc, putw — put character or word on a stream

## SYNOPSIS

```
#include <stdio.h>
```

```
int putc (c, stream)
```

```
int c;
```

```
FILE *stream;
```

```
int putchar (c)
```

```
int c;
```

```
int fputc (c, stream)
```

```
int c;
```

```
FILE *stream;
```

```
int putw (w, stream)
```

```
int w;
```

```
FILE *stream;
```

## DESCRIPTION

*Putc* writes the character *c* onto the output *stream* (at the position where the file pointer, if defined, is pointing). *Putchar(c)* is defined as *putc(c, stdout)*. *Putc* and *putchar* are macros.

*Fputc* behaves like *putc*, but is a function rather than a macro. *Fputc* runs more slowly than *putc*, but it takes less space per invocation and its name can be passed as an argument to a function.

*Putw* writes the word (i.e., integer) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. *Putw* neither assumes nor causes special alignment in the file.

## RETURN VALUE

On success, *putc*, *fputc*, and *putchar* each return the value they have written. On failure, they return the constant EOF. This will occur if the file *stream* is not open for writing or if the output file cannot be grown. The function *putw* returns non-zero when an error has occurred; otherwise the function returns 0.

## SEE ALSO

*fclose*(3S), *ferror*(3S), *fopen*(3S), *fread*(3S), *printf*(3S), *puts*(3S), *setbuf*(3C), *stdio*(3S).

## APPLICATION USAGE

Because it is implemented as a macro, *putc*(3S) treats incorrectly a *stream* argument with side effects. In particular, *putc(c, \*f++)*; doesn't work sensibly. *Fputc*(3S) should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw*(3S) are machine-dependent, and may not be read using *getw*(3S) on a different processor.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

### Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following changes:

A paragraph on buffered output has been moved to the *stdio(3S)* entry.

The **RETURN VALUE** section has been rewritten.

## NAME

putenv — change or add value to environment

## SYNOPSIS

```
int putenv (string)
char *string;
```

## DESCRIPTION

*String* points to a string of the form "*name=value*". *Putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

## RETURN VALUE

*Putenv* returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero (0).

## SEE ALSO

*exec*(2), *getenv*(3C), *malloc*(3X).

## APPLICATION USAGE

*Putenv* manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*(3C). However, *envp* (the third argument to *main*) is not changed.

This routine uses *malloc*(3X) to enlarge the environment.

After *putenv* is called, environmental variables are not in alphabetical order.

A potential error is to call *putenv* with an automatic variable as the argument, then return from the calling function while *string* is still part of the environment.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The addition of the statement in the APPLICATIONS USAGE section: "After *putenv* is called, environment variables are not in alphabetical order".





## NAME

putpwent — write password file entry

## SYNOPSIS

```
#include <stdio.h>
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

## DESCRIPTION

*Putpwent* is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure created by *getpwent*(3C) (or *getpwent*(3C) or *getpwnam*(3C)), *putpwent* writes a line on the stream *f* which matches the format of */etc/passwd*.

## RETURN VALUE

*Putpwent* returns non-zero if an error was detected during its operation, otherwise zero (0).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

The call for inclusion of `<stdio.h>` has been added.





## NAME

puts, fputs — put a string on a stream

## SYNOPSIS

```
#include <stdio.h>
```

```
int puts (s)
```

```
char *s;
```

```
int fputs (s, stream)
```

```
char *s;
```

```
FILE *stream;
```

## DESCRIPTION

*Puts* writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream *stdout*.

*Fputs* writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

## RETURN VALUE

Both routines return EOF on error. This will happen if the routines try to write on a file that has not been opened for writing.

## SEE ALSO

error(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

## APPLICATION USAGE

*Puts* appends a new-line character while *fputs* does not.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.



## NAME

qsort — quicker sort

## SYNOPSIS

```
void qsort (base, nel, width, compar)
char *base;
unsigned nel, width;
int (*compar)();
```

## DESCRIPTION

*Qsort* is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

*Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Width* is the size of an element in bytes. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. As the function must return an integer less than, equal to, or greater than zero, so must the first argument to be considered be less than, equal to, or greater than the second.

## SEE ALSO

bsearch(3C), lsearch(3C), string(3C).

## APPLICATION USAGE

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The order in the output of two items which compare as equal is unpredictable.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

*rand*, *srand* — simple random-number generator

## SYNOPSIS

```
int rand ()  
  
void srand (seed)  
unsigned seed;
```

## DESCRIPTION

*Rand* uses a multiplicative congruential random-number generator with period  $2^{32}$  that returns successive pseudo-random numbers in the range from 0 to  $2^{15}-1$ .

*Srand* can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of one (1).

## SEE ALSO

*drand48*(3C).

## APPLICATION USAGE

*Drand48*(3C) provides a much more elaborate random number generator.

The following functions define the semantics of *rand* and *srand*.

```
static unsigned long int next = 1;  
int rand()  
{  
    next = next * 1103515245 + 12345;  
    return((unsigned int)(next/65536) % 32768);  
}  
void srand(seed)  
unsigned int seed;  
{  
    next = seed;  
}
```

Specifying the semantic makes it possible to reproduce the behavior of programs that use pseudo-random sequences. This facilitates the testing of portable applications in different implementation.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





NAME

regcmp — (WITHDRAWN)

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID.

Issue 2

Previously, *regcmp*(3X) was included in Issue 1 in error. The functionality described on this page has been replaced by that described in *regexp*(3X). Issue 2 of the SVID makes the same change.



## NAME

regex — regular-expression compile and match routines

## SYNOPSIS

```
#define INIT declarations
#define GETC() getc code
#define PEEK() peek code
#define UNGETC() ungetc code
#define RETURN(ptr) return code
#define ERROR(val) error code

#include <regex.h>

char *compile(instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step(string, expbuf)
char *string, *endbuf;

advance(string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;
```

## DESCRIPTION

These functions are general-purpose regular-expression matching routines to be used in programs that perform regular-expression matching. These functions are defined by the `<regex.h>` header file.

The functions *step* and *advance* do pattern matching given a character string and a compiled regular-expression as input.

The function *compile* takes as input a regular-expression as defined below and produces a compiled expression that can be used with *step* or *advance*.

A regular-expression, *re*, specifies a set of character strings. A member of this set of strings is said to be *matched* by the *re*. Some characters have special meaning when used in an *re*: other characters stand for themselves.

The regular-expressions available for use with the function *regex* are constructed as follows:

Expression	Meaning
<i>c</i>	the character <i>c</i> where <i>c</i> is not a special character.
<code>\c</code>	The character <i>c</i> where <i>c</i> is any character, except a digit in the range 1 — 9.
<code>^</code>	the beginning of the line being compared.
<code>\$</code>	The end of the line being compared.
<code>.</code>	any character in the input.
<code>[s]</code>	any character in the set <i>s</i> , where <i>s</i> is a sequence of characters and/or a range of characters, e.g., <code>[c-c]</code> .
<code>[s]</code>	any character not in the set <i>s</i> , where <i>s</i> is defined as above.



- $r^*$  zero or more successive occurrences of the regular-expression  $r$ . The longest match is chosen.
- $rx$  the occurrence of regular-expression  $r$  followed by the occurrence of regular-expression  $x$ . (Concatenation)
- $r\{m,n\}$  any number of  $m$  through  $n$  successive occurrences of the regular-expression  $r$ . The regular expression  $r\{m\}$  matches exactly  $m$  occurrences  $r\{m,\}$  matches at least  $m$  occurrences.
- $\backslash(r\backslash)$  the regular expression  $r$ . When  $\backslash n$  (where  $n$  is a number greater than zero) appears in a constructed regular-expression, it stands for the regular-expression  $x$  where  $x$  is the  $n^{\text{th}}$  regular-expression enclosed in  $\backslash($  and  $\backslash)$  strings that appeared earlier in the constructed regular-expression. For example,  $\backslash(r\backslash)x\backslash(y\backslash)z\backslash 2$  is the concatenation of regular-expressions  $rxzy$ .

Characters that have special meaning except where they appear within square brackets,  $[ ]$ , or are preceded by  $\backslash$  are:  $.$ ,  $*$ ,  $[$ ,  $\backslash$ . Other special characters, such as  $\$$  have special meaning in more restricted contexts.

The character  $\wedge$  at the beginning of an expression permits a successful match only immediately after a new-line, and the character  $\$$  at the end of an expression requires a trailing new-line.

Two characters have special meaning only when used within square brackets. The character  $-$  denotes a range,  $[c-c]$ , unless it is just after the open bracket or before the closing bracket,  $[-c]$  or  $[c-]$  in which case it has no special meaning. When used within brackets, the character  $\wedge$  has the meaning *complement* of if it immediately follows the open bracket,  $[c]$ , elsewhere between brackets,  $[c]$ , it stands for the ordinary character  $\wedge$ .

The special meaning of the  $\backslash$  operator can be escaped *only* by preceding it with another  $\backslash$ , e.g.  $\backslash \backslash$ .

Programs must have the following five macros declared before the `#include <regexp.h>` statement. These macros are used by the *compile* routine. The macros `GETC()`, `PEEK()` and `UNGETC()` operate on the regular-expression given as input to *compile*.

- `GETC()` This macro returns the value of the next character in the regular-expression pattern. Successive calls to `GETC()` should return successive characters of the regular-expression.
- `PEEK()` This macro returns the next character in the regular-expression. Immediately successive calls to `PEEK()` should return the same character, which should also be the next character returned by `GETC()`.
- `UNGETC(c)` This macro causes the argument  $c$  to be returned by the next call to `GETC()` and `PEEK()`. No more than one character of pushback is ever needed and this character is guaranteed to be the last character read by `GETC()`. The value of the macro `UNGETC(c)` is always ignored.
- `RETURN(ptr)` This macro is used on normal exit of the *compile* routine. The value of the argument  $ptr$  is a pointer to the character after the last character of the compiled regular-expression. This is useful to

programs which have memory allocation to manage.

*ERROR(val)* This macro is the abnormal return from the *compile* routine. The argument *val* is an error number [see **ERRORS** below for meanings]. This call should never return.

The syntax of the *compile* routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the *INIT* declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of `((char*)0)` for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular-expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular-expression may be placed. If the compiled expression cannot fit in (*endbuf-expbuf*) bytes, a call to *ERROR(50)* is made.

The parameter *eof* is the character which marks the end of the regular-expression. For example, in *ed(1)*, this character is usually a `/`.

Each program that includes the `<regexp.h>` header file must have a `#define` statement for *INIT*. It is used for dependent declarations and initialisations. Most often it is used to set a register variable to point to the beginning of the regular-expression so that this register variable can be used in the declarations for *GETC()*, *PEEKC()* and *UNGETC()*. Otherwise it can be used to declare external variables that might be used by *GETC()*, *PEEKC()* and *UNGETC()*. See **EXAMPLES** below.

The first parameter to the *step* function is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter, *expbuf*, is the compiled regular-expression which was obtained by a call to the function *compile*.

The function *step* returns non-zero if some sub-string of *string* matches the regular-expression in *expbuf* and zero if there is no match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable *loc1* points to the first character that matched the regular-expression; the variable *loc2* points to the character after the last character that matches the regular-expression. Thus if the regular-expression matches the entire input string, *loc1* will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

The function *advance* returns non-zero if the initial substring of *string* matches the regular-expression in *expbuf*. If there is a match an external character pointer, *loc2*, is set as a side effect. The variable *loc2* points to the next character in *string* after the last character that matched.



When *advance* encounters a `*` or `\{ \}` sequence in the regular-expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular-expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the `*` or `\{ \}`. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at some time during the backing up process, *advance* will break out of the loop that backs up and will return zero.

The external variables *circf*, *sed*, and *nbra* are reserved.

### RETURN VALUE

The function *compile* uses the macro *RETURN()* on success and the macro *ERROR()* on failure, see above. The functions *step* and *advance* return non-zero on a successful match and zero if there is no match.

### ERRORS

- 11 range endpoint too large.
- 16 bad number.
- 25 `\digit` out of range.
- 36 illegal or missing delimiter.
- 41 no remembered search string.
- 42 `\( \)` imbalance.
- 43 too many `\(` .
- 44 more than two numbers given in `\{ \}` .
- 45 `}` expected after `\` .
- 46 first number exceeds second in `\{ \}` .
- 49 `[ ]` imbalance.
- 50 regular-expression overflow.

### EXAMPLES

The following is an example of how the regular-expression macros and calls might be defined by an application program:

```
#define INIT                register char *sp = instring;
#define GETC()              (*sp)
#define PEEKC()             (*sp)
#define UNGETC(c)           (--sp)
#define RETURN(c)           return;
#define ERROR(c)            regerr()

#include <regexp.h>

...
(void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
if (step(linebuf, expbuf) )
    succeed();
```



**CHANGE HISTORY**

First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID with the following changes:

The parameter *c* was added to the description of the *UNGETC* macro.

In the description of the parameter *eof*, the example has been taken from UNIX System V, Release 2.0.



## NAME

scanf, fscanf, sscanf — convert formatted input (NLS)

## SYNOPSIS

```
#include <stdio.h>
```

```
int scanf (format [ , pointer ] ... )
char *format;
```

```
int fscanf (stream, format [ , pointer ] ... )
FILE *stream;
char *format;
```

```
int sscanf (s, format [ , pointer ] ... )
char *s, *format;
```

## DESCRIPTION

*Scanf* reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not '%'), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character '%', an optional assignment suppressing character '\*', an optional numerical maximum field width, an optional 'l' (ell) or 'h' indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by '\*'. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except 'l' and 'c', white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

- % a single '%' is expected in the input at this point; no assignment is done.
- d a decimal integer is expected; the corresponding argument should be an integer pointer.
- u an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
- o an octal integer is expected; the corresponding argument should be an integer pointer.



- x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
- e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a **float**. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an 'E' or an 'e', followed by an optional '+', '-', or space, followed by an integer.
- s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added automatically. The input field is terminated by a white-space character.
- c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use "%1s". If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.
- [ indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first-last*, thus "[0123456789]" may be expressed "[0-9]". Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating '\0', which will be added automatically. At least one character must match for this conversion to be considered successful.

If an invalid conversion character follows the '%', the results of the operation may not be predictable.

The conversion characters 'd', 'u', 'o', and 'x' may be preceded by 'l' or 'h' to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters 'e', 'f', and 'g' may be preceded by 'l' to indicate that a pointer to **double** rather than to **float** is in the argument list. The 'l' or 'h' modifier is ignored for other conversion characters.

*Scanf* conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

*Scanf* returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

## RETURN VALUE

These functions return EOF on end of input and a short count for missing or illegal data items.

## SEE ALSO

getc(3S), printf(3S), strtod(3C), strtol(3C).

## FUTURE DIRECTIONS

*Scanf* will make available character string representations for  $\infty$  and "Not a Number" (NaN: a symbolic entity encoded in floating point format) to support the IEEE P754 standard.

## APPLICATION USAGE

Trailing white space (including a new-line) is left unread unless matched in the control string.

The success of literal matches and suppressed assignments is not directly determinable.

## EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain "thompson \0". Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip 0123, and place the string "56\0" in *name*. The next call to *getchar* (see *getc*(3S)) will return 'a'.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

The descriptions of the *i* and *n* conversion codes have been deleted.





## NAME

setbuf, setvbuf — assign buffering to a stream

## SYNOPSIS

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;
```

## DESCRIPTION

*Setbuf* may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer input/output will be completely unbuffered.

A constant BUFSIZ, defined in the <stdio.h> header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

*Setvbuf* may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in <stdio.h>) are:

- \_IOFBF causes input/output to be fully buffered.
- \_IOLBF causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.
- \_IONBF causes input/output to be completely unbuffered.

If *buf* is not the NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *Size* specifies the size of the buffer to be used. The constant BUFSIZ in <stdio.h> is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line buffered and all other input/output is fully buffered.

## RETURN VALUE

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

## SEE ALSO

fopen(3S), getc(3S), malloc(3X), putc(3S), stdio(3S).

## APPLICATION USAGE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.



## NAME

setjmp, longjmp — non-local goto

## SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;
```

## DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

*Setjmp* saves its stack environment in *env* (whose type, *jmp\_buf*, is defined in the *<setjmp.h>* header file) for later use by *longjmp*. It returns the value zero (0).

*Longjmp* restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (the caller of which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data have values as of the time *longjmp* was called.

## SEE ALSO

signal(2).

## APPLICATION USAGE

If *longjmp* is called even though *env* was never primed by a call to *setjmp*, or when the last such call was in a function which has since returned, the behaviour is undefined.

If the call to *longjmp* is in a different function from the corresponding call to *setjmp*, variables with **register** storage class may have unpredictable values.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

*sinh*, *cosh*, *tanh* — hyperbolic functions (OPTIONAL)

## SYNOPSIS

```
#include <math.h>
```

```
double sinh (x)
```

```
double x;
```

```
double cosh (x)
```

```
double x;
```

```
double tanh (x)
```

```
double x;
```

## DESCRIPTION

*Sinh*, *cosh*, and *tanh* return, respectively, the hyperbolic sine, cosine and tangent of their argument.

## RETURN VALUE

*Sinh* and *cosh* return HUGE (and *sinh* may return —HUGE for negative *x*) when the correct value would overflow and set *errno* to [ERANGE].

These error-handling procedures may be changed with the function *matherr*(3M).

## SEE ALSO

*matherr*(3M).

## FUTURE DIRECTIONS

A macro HUGE\_VAL will be defined in the header file <math.h>. This macro will call a function which will either return  $+\infty$  on a system supporting IEEE P754 standard or +(MAXDOUBLE) on a system that does not support the IEEE P754 standard.

*Sinh* and *cosh* will return HUGE\_VAL (*sinh* will return —HUGE\_VAL for negative *n*) when the correct value overflows.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID. The *optional* mathematical group is mandatory in SVID.





## NAME

sleep — suspend execution for interval

## SYNOPSIS

unsigned sleep (seconds)  
unsigned seconds;

## DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed intervals, and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. But if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

## SEE ALSO

alarm(2), pause(2), signal(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The words "1-second" have been deleted from after "fixed" in the second sentence of the DESCRIPTION section.



## NAME

ssignal, gsignal — software signals

## SYNOPSIS

```
#include <signal.h>

int (*ssignal (sig, action))()
int sig, (*action)();

int gsignal (sig)
int sig;
```

## DESCRIPTION

*Ssignal* and *gsignal* implement a software facility similar to *signal(2)*. This facility is made available to users for their own purposes.

Software signals made available to users are listed in *signal(2)*. A call to *ssignal* associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to *gsignal*. Raising a software signal causes the action established for that signal to be taken.

The first argument to *ssignal* is a signal listed in *signal(2)* for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action* function or one of the manifest constants SIG\_DFL (default) or SIG\_IGN (ignore). *Ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns SIG\_DFL.

*Gsignal* raises the signal identified by its argument, *sig*:

If an action function has been established for *sig*, then that action is reset to SIG\_DFL and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.

If the action for *sig* is SIG\_IGN, *gsignal* returns the value 1 and takes no other action.

If the action for *sig* is SIG\_DFL, *gsignal* returns the value zero (0) and takes no other action.

If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

## SEE ALSO

signal(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

stdio — standard buffered input/output package

## SYNOPSIS

```
#include <stdio.h>
```

```
FILE *stdin, *stdout, *stderr;
```

## DESCRIPTION

The functions described as Standard I/O routines (*stdio*) constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type *FILE*. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the *<stdio.h>* header file and associated with the standard open files:

<i>stdin</i>	standard input file
<i>stdout</i>	standard output file
<i>stderr</i>	standard error file

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen* (see *fopen*(3S)) will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). *Setbuf*(3C) or *setvbuf*(3C) may be used to change the stream's buffering strategy.

A constant *NULL* designates a nonexistent pointer.

An integer-constant *EOF* is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant *BUFSIZ* specifies the size of the buffers used by the particular implementation.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The Standard I/O related functions and constants are declared in that header file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *ferror*, *feof*, *clearerr*, and *fileno*.

## RETURN VALUE

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

### SEE ALSO

open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(3S), fclose(3S),  
ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S),  
printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3C), system(3S), tmpfile(3S),  
tmpnam(3S), ungetc(3S).

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID.

#### Issue 2

This entry moved to the introduction of Issue 2 of the SVID. It remains here in the X/OPEN description.

The paragraph on buffered output streams has been moved here from the *putc(3S)* entry.



## NAME

*strcat*, *strncat*, *strcmp*, *strncmp*, *strcpy*, *strncpy*, *strlen*, *strchr*, *strrchr*, *strpbrk*, *strspn*, *strcspn*, *strtok* — string operations (NLS)

## SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

## DESCRIPTION

The arguments *s1*, *s2* and *s* point to strings (arrays of characters terminated by a NULL character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

*Strcat* appends a copy of string *s2* to the end of string *s1*. *Strncat* appends at most *n* characters. Each returns a pointer to the null-terminated result.

*Strcmp* compares its arguments and returns an integer less than, equal to, or greater than zero (0), according as *s1* is lexicographically less than, equal to, or greater than *s2*. *Strncmp* makes the same comparison but looks at at most *n* characters.

*Strcpy* copies string *s2* to *s1*, stopping after the NULL character has been copied. *Strncpy* copies exactly *n* characters, truncating *s2* or adding NULL characters to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

*Strlen* returns the number of characters in *s*, not including the terminating NULL character.

*Strchr* (*strchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. The NULL character terminating a string is considered to be part of the string.

*Strpbrk* returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

*Strspn* (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

*Strtok* considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string *s1* immediately following that token. In this way subsequent calls will work through the string *s1*, returning a pointer to the first character of each subsequent token. A NULL character will have been written into *s1* by *strtok* immediately following the token. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

### SEE ALSO

memory(3C).

### APPLICATION USAGE

All these functions are declared in the `<string.h>` header file.

*Strcmp* and *strncmp* use native character comparison. The sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

### FUTURE DIRECTIONS

The type of the argument *n* to *strncat*, *strncmp* and *strncpy* and the type of the value returned by *strlen* will be declared through the `typedef` facility in a header file as `size_t`.

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID.

Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

The paragraph describing *strtok* has been clarified.





## NAME

strtod, atof — convert string to double-precision number (NLS)

## SYNOPSIS

```
double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;
```

## DESCRIPTION

*Strtod* returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

*Strtod* recognizes an optional string of "white-space" characters (as defined by *isspace* in *ctype*(3C)), then an optional sign, then a string of digits optionally containing a decimal point, then an optional *e* or *E* followed by an optional sign, followed by an integer.

If the value of *ptr* is not *(char \*\*)0*, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, *\*ptr* is set to *str*, and zero is returned.

*Atof*(*str*) is equivalent to *strtod*(*str*, *(char \*\*)0*).

## RETURN VALUE

If the correct value would cause overflow,  $\pm$ HUGE is returned (according to the sign of the value), and *errno* is set to [ERANGE].

If the correct value would cause underflow, zero is returned and *errno* is set to [ERANGE].

## SEE ALSO

*ctype*(3C), *scanf*(3S), *strtol*(3C).

## FUTURE DIRECTIONS

A macro HUGE\_VAL will be defined in the header file *<math.h>*. This macro will call a function which will either return  $+\infty$  on a system supporting IEEE P754 standard or  $+\{\text{MAXDOUBLE}\}$  on a system that does not support the IEEE P754 standard.

If the correct value overflows, *strtod* will return  $\pm$ HUGE\_VAL (according to the sign of the value).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.





## NAME

*strtol*, *atol*, *atoi* — convert string to integer

## SYNOPSIS

```
long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;
```

## DESCRIPTION

*Strtol* returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by *isspace* in *ctype*(3C)) are ignored.

If the value of *ptr* is not *(char \*\*)0*, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment or by an explicit cast.

*Atol(str)* is equivalent to *strtol(str, (char \*\*)0, 10)*.

*Atoi(str)* is equivalent to *(int) strtol(str, (char \*\*)0, 10)*.

## SEE ALSO

*ctype*(3C), *scanf*(3S), *strtod*(3C).

## APPLICATION USAGE

Overflow conditions are ignored.

## FUTURE DIRECTIONS

Error handling will be added to *strtol*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.



NAME

swab — swap bytes

SYNOPSIS

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

DESCRIPTION

*Swab* copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*—1 instead. If *nbytes* is negative, *swab* does nothing.

CHANGE HISTORY

Issue 1.

Derived from the entry in Issue 1 of the SVID.





## NAME

system — issue a command

## SYNOPSIS

```
#include <stdio.h>
```

```
int system (string)
```

```
char *string;
```

## DESCRIPTION

*System* causes the *string* to be given to a command interpreter and execution process as input.

## Definitions

A *blank* is a tab or a space. A *parameter name* is a sequence of letters, digits, or underscores beginning with a letter or underscore. A *parameter* is either a parameter name or any of the characters *?*, *\$*, or *!*.

## Commands

A *simple-command* is a sequence of non-blank *words* separated by *blanks*. The first word specifies the pathname or file name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0, see *exec(2)*. The *value* of a *simple-command* is its exit status if it terminates normally, or (octal) 0200+*status* if it terminates abnormally (see *signal(2)* for a list of status values).

A *pipeline* is a sequence of two or more *commands* separated by *|*. The standard output of each command but the last is connected by a *pipe(2)* to the standard input of the next command. Each command is run as a separate process; the command execution process waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more simple commands or pipelines separated by *;*, *&*, *&&*, or *| |*. A list may optionally be terminated by *;* or *&*. Of these four symbols, *;* and *&* have equal precedence, which is lower than that of *&&* and *| |*. The symbols *&&* and *| |* also have equal precedence.

A semicolon (*;*) causes sequential execution of the preceding pipeline;

An ampersand (*&*) causes asynchronous execution of the preceding pipeline;

The symbol *&&* (*| |*) causes the *list* following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrarily long sequence of new-lines may appear in a *list*, instead of a semicolon, to delimit commands.

A *command* is either a *simple-command* or one of the following. Unless otherwise stated, the value returned by a command is that of the last *simple-command* executed in the command list.

## Comments

A word beginning with *#* causes that word and all the following characters up to a new-line to be ignored.

## Commr

The standard output from a command enclosed in a pair of grave accents (*``*) may be used as part or all of a word; trailing new-lines are removed.

### Parameter Substitution

The character \$ is used to introduce substitutable *keyword parameters*. Keyword parameters (also known as variables) may be assigned values by writing:

```
parameter-name=value
```

```
parameter-name=value
```

```
....
```

`${parameter}`

The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name.

The following parameters are automatically set:

? The decimal value returned by the last synchronously executed command.

\$ The process number of this process.

! The process number of the last background command invoked.

The following parameters are used by the command execution process:

HOME The initial working (home) directory, initially set from the 6th field in the */etc/passwd* file (see *passwd(4)*).

PATH The search path for commands (see *Execution* below).

### Blank Interpretation

After parameter and command substitution, the results of substitution are scanned for internal field separator characters (space, tab and newline) and split into distinct arguments where such characters are found. Explicit null arguments ("" or `?`) are retained. Implicit null arguments (those resulting from *parameters* that have no values) are removed.

### File Name Generation

Following substitution, each command *word* is scanned for the characters \*, ?, and [. If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character . at the start of a file name or immediately following a /, as well as the character / itself, must be matched explicitly.

\* Matches any string, including the null string.

? Matches any single character.

[...] Matches any one of the enclosed characters. A pair of characters separated by — matches any character lexically between the pair, inclusive. If the first character following the opening '[' is a "!" any character not enclosed is matched.

### Quoting

The following characters have a special meaning and cause termination of a word unless quoted:

```
; & ( ) | ^ < > new-line space tab
```

A character may be *quoted* (i.e., made to stand for itself) by preceding it with a \. The pair `\new-line` is ignored. All characters enclosed between a pair of single quote marks ('), except a single quote, are quoted. Inside double quote marks (""), parameter and command substitution occurs and \ quotes the characters \, ', and \$.



## Input/Output

Before a command is executed, its input and output may be redirected using a special notation. The following may appear anywhere in a *simple-command* or may precede or follow a *command* and are *not* passed on to the invoked command; substitution occurs before *word* or *digit* is used:

< <i>word</i>	Use file <i>word</i> as standard input (file descriptor 0).
> <i>word</i>	Use file <i>word</i> as standard output (file descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length.
>> <i>word</i>	Use file <i>word</i> as standard output. If the file exists output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.
<& <i>digit</i>	Use the file associated with file descriptor <i>digit</i> as standard input. Similarly for the standard output using >& <i>digit</i> .
<&-	The standard input is closed. Similarly for the standard output using >&-.

If any of the above is preceded by a digit, the file descriptor which will be associated with the file is that specified by the digit (instead of the default 0 or 1). For example:

```
... 2>&1
```

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

```
... 1>xxx 2>&1
```

first associates file descriptor 1 with file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (i.e., *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

If a command is followed by & the default standard input for the command is the empty file */dev/null*. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking process as modified by input/output specifications.

## Environment

The *environment* (see *exec(2)*) is a list of parameter name-value pairs that is passed to an executed program in the same way as a normal argument list. On invocation, the environment is scanned and a parameter is created for each name found, giving it the corresponding value.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to parameters.

```
TERM=450 cmd
```

## Signals

The **SIGINT** and **SIGQUIT** signals for an invoked command are ignored if the command is followed by &; otherwise signals have the values inherited by the command execution process from its parent.

## Execution

Each time a command is executed, the above substitutions are carried out. A new process is created and an attempt is made to execute the command via *exec(2)*.

The parameter `PATH` defines the search path for the directory containing the command. Alternative directory names are separated by a colon (:). The default path is `:/bin:/usr/bin` (specifying the current directory, `/bin`, and `/usr/bin`, in that order). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If the command name contains a `/` the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an `a.out` file, it is assumed to be a file containing commands.

Conventionally, `system` has been implemented with the Bourne shell, `sh(1)`. The current definition of `system` is not intended to preclude that or its implementation by another command interpreter that provides the minimum functionality described here. Of course, any implementation may provide a superset of the functionality described.

#### RETURN VALUE

Errors, such as syntax errors, cause a non-zero return value and execution of the command file is abandoned. Otherwise, the exit status of the last command executed is returned.

#### FILES

`/dev/null`  
`/etc/passwd`

#### SEE ALSO

`dup(2)`, `exec(2)`, `fork(2)`, `pipe(2)`, `signal(2)`, `ulimit(2)`, `umask(2)`, `wait(2)`, `passwd(4)`.

#### CHANGE HISTORY

##### Issue 1

Derived from the entry in Issue 1 of the SVID.

##### Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

All text referring to positional parameters, here documents and the parameter `$-` has been removed.



## NAME

tmpfile — create a temporary file

## SYNOPSIS

```
#include <stdio.h>
```

```
FILE *tmpfile ()
```

## DESCRIPTION

*Tmpfile* creates a temporary file using a name generated by *tmpnam*(3S), and returns a corresponding *FILE* pointer. If the file cannot be opened, an error message is printed and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

## RETURN VALUE

If the file cannot be opened, an error message is printed and a NULL pointer is returned.

## SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), tmpnam(3S).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The RETURN VALUE sentence has also been repeated in the DESCRIPTION section.





## NAME

*tmpnam*, *tempnam* — create a name for a temporary file

## SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

## DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

*Tmpnam* always generates a file name using the path-prefix defined as {P\_tmpdir} in the <stdio.h> header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least {L\_tmpnam} bytes, where {L\_tmpnam} is a constant defined in <stdio.h>; *tmpnam* places its result in that array and returns *s*.

*Tempnam* allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a name for an appropriate directory, the path-prefix defined as {P\_tmpdir} in the <stdio.h> header file is used. If that directory is not accessible, /tmp will be used as a last resort.

*Tempnam* uses *malloc*(3X) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* (see *malloc*(3X)). If *tempnam* cannot return the expected result for any reason, i.e. *malloc*(3X) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

## SEE ALSO

*creat*(2), *unlink*(2), *fopen*(3S), *malloc*(3X), *mktemp*(3C), *tmpfile*(3S).

## APPLICATION USAGE

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

The functions *tmpnam* and *tempnam* generate a different file name each time they are called.

Files created using these functions and either *fopen*(3S) or *creat*(2) are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *fclose*(2) or *unlink*(2) to remove the file when its use is ended.

If called more than {TMP\_MAX} times in a single process, these functions will start recycling previously used names. Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

**CHANGE HISTORY**  
**Issue 1**

Derived from the entry in Issue 1 of the SVID.



## NAME

*sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *atan2* — trigonometric functions (OPTIONAL)

## SYNOPSIS

```
#include <math.h>
```

```
double sin (x)
```

```
double x;
```

```
double cos (x)
```

```
double x;
```

```
double tan (x)
```

```
double x;
```

```
double asin (x)
```

```
double x;
```

```
double acos (x)
```

```
double x;
```

```
double atan (x)
```

```
double x;
```

```
double atan2 (y, x)
```

```
double y, x;
```

## DESCRIPTION

*Sin*, *cos* and *tan* return respectively the sine, cosine and tangent of their argument *x*, measured in radians.

*Asin* returns the arcsine of *x*, in the range  $-\pi/2$  to  $\pi/2$ .

*Acos* returns the arccosine of *x*, in the range 0 to  $\pi$ .

*Atan* returns the arctangent of *x*, in the range  $-\pi/2$  to  $\pi/2$ .

*Atan2* returns the arctangent of *y/x*, in the range  $-\pi$  to  $\pi$ , using the signs of both arguments to determine the quadrant of the return value.

## RETURN VALUE

*Sin*, *cos*, and *tan* lose accuracy when their argument is far from zero (0). For arguments sufficiently large, these functions return zero when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to [ERANGE].

If the magnitude of the argument of *asin* or *acos* is greater than one, or if both arguments of *atan2* are zero, zero is returned and *errno* is set to [EDOM]. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

## SEE ALSO

*matherr*(3M).

### CHANGE HISTORY

#### Issue 1

Derived from the entry in Issue 1 of the SVID.

The *optional* mathematical group is mandatory in SVID.

## NAME

*tsearch*, *tfind*, *tdelete*, *twalk* — manage binary search trees

## SYNOPSIS

```
#include <search.h>

char *tsearch (key, rootp, compar)
char *key;
char **rootp;
int (*compar)();

char *tfind (key, rootp, compar)
char *key;
char **rootp;
int (*compar)();

char *tdelete (key, rootp, compar)
char *key;
char **rootp;
int (*compar)();

void twalk (root, action)
char *root;
void (*action)();
```

## DESCRIPTION

*Tsearch*, *tfind*, *tdelete*, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than zero (0), according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

*Tsearch* is used to build and access the tree. *Key* is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *\*key* (the value pointed to by *key*), a pointer to this found datum is returned. Otherwise, *\*key* is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. *Rootp* points to a variable that points to the root of the tree. A NULL value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

*Tdelete* deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

*Twalk* traverses a binary search tree. *Root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited.



The second argument is a value from an enumeration data type

```
typedef enum { preorder, postorder, endorder, leaf } VISIT
```

(defined in the `<search.h>` header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

### RETURN VALUE

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.

A NULL pointer is returned by *tsearch*, *tfind* and *tdelete* if *rootp* is NULL on entry.

If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

### SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C), search(5).

### APPLICATION USAGE

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

The *root* argument to *twalk* is one level of indirection less than the *rootp* arguments to *tsearch* and *tdelete*.

There are two nomenclatures used to refer to the order in which tree nodes are visited. *Tsearch* uses preorder, postorder and endorder to respectively refer to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

If the calling function alters the pointer to the root, results are unpredictable.

### EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <search.h>
#include <stdio.h>

struct node {          /* pointers to these are stored in the tree */
    char *string;
    int length;
};
char string_space[10000]; /* space to store strings */
struct node nodes[500];   /* nodes to store */
struct node *root = NULL; /* this points to the root */

main( )
{
    char *strptr = string_space;
    struct node *nodeptr = nodes;
    void print_node( ), twalk( );
    int i = 0, node_compare( );
```

```

while (gets(strptr) != NULL && i++ < 500) {
    /* set node */
    nodeptr->string = strptr;
    nodeptr->length = strlen(strptr);
    /* put node into the tree */
    (void) tsearch((char *)nodeptr, &root,
        node_compare);
    /* adjust pointers, so we don't overwrite tree */
    strptr += nodeptr->length + 1;
    nodeptr++;
}
twalk(root, print_node);
}
/*

This routine compares two nodes, based on an
alphabetical ordering of the string field.

*/
int
node_compare(node1, node2)
char *node1, *node2;
{
    return strcmp((struct node *) node1->string,
        (struct node *) node2->string);
}
/*

This routine prints out a node, the first time
twalk encounters it.

*/

void
print_node(node, order, level)
struct node **node;
VISIT order;
int level;
{
    if (order == preorder || order == leaf) {
        (void)printf("string = %20s, length = %d\n",
            (*node)->string, (*node)->length);
    }
}

```

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

A programming error in the *node\_compare* part of the example has been corrected. The SVID reads:

```
return strcmp(node1->string, node2->string);
```





## NAME

*ttyname*, *isatty* — find name of a terminal

## SYNOPSIS

```
char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;
```

## DESCRIPTION

*Ttyname* returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

*Isatty* returns 1 if *fildes* is associated with a terminal device, zero (0) otherwise.

## FILES

/dev/\*

## RETURN VALUE

*Ttyname* returns a NULL pointer if *fildes* does not describe a terminal device in directory */dev*.

## APPLICATION USAGE

The return value points to static data whose content is overwritten by each call.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.



NAME

ttyslot — find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

*Ttyslot* returns the index of the current user's entry in the */etc/utmp* file.

FILES

*/etc/utmp*

SEE ALSO

getut(3C), ttyname(3C).

RETURN VALUE

A value of zero (0) is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

CHANGE HISTORY

Issue 1

This function was not included in Issue 1 of the SVID. It was taken from UNIX System V Release 2.0.

Issue 2

In Issue 2, the title **DIAGNOSTICS** has been corrected to **RETURN VALUE** .





## NAME

`ungetc` — push character back into input stream

## SYNOPSIS

```
#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;
```

## DESCRIPTION

*Ungetc* inserts the character *c* into the buffer associated with an input *stream*. That character *c*, will be returned by the next *getc*(3S) call on that *stream*. *Ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered. In the case that *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

If *c* equals EOF *ungetc* does nothing to the buffer and returns EOF.

*Fseek*(3S) erases all memory of inserted characters.

## RETURN VALUE

*Ungetc* returns EOF if it cannot insert the character.

## SEE ALSO

*fseek*(3S), *getc*(3S), *setbuf*(3C).

## APPLICATION USAGE

The use of pushback without a prior *read* in the case of *stdin* is not recommended as this functionality has been removed in Issue 2 of the SVID. It may be subject to withdrawal later.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

## Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following change:

The APPLICATION USAGE has been added.





## NAME

`vprintf`, `vfprintf`, `vsprintf` — print formatted output of a `varargs` argument list

## SYNOPSIS

```
#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int fprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

## DESCRIPTION

`vprintf`, `vfprintf`, and `vsprintf` are the same as `printf`, `fprintf`, and `sprintf` respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `<varargs.h>`.

## SEE ALSO

`printf(3S)`.

## EXAMPLE

The following demonstrates how `vfprintf` could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>

/*
 *      error should be called like
 *      error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 *      separately declared because of the definition of varargs.
 */
va_dcl
{
    va_list args;
    char *fmt;

    va_start(args);
    /* print out name of function causing error */
    (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
    fmt = va_arg(args, char *);
    /* print out remainder of message */
```

```
(void)vfprintf(fmt, args);  
va_end(args);  
(void)abort( );
```

```
}
```

## CHANGE HISTORY

### Issue 1

Derived from the entry in Issue 1 of the SVID.



## *Chapter 4*

# ***File Formats***

This chapter outlines the formats of various files.





## NAME

acct — per-process accounting file format

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/acct.h>
```

## DESCRIPTION

Files produced as a result of calling *acct(2)* have records in the form defined by *<sys/acct.h>*, whose contents include the following declarations and definitions.

A typedef definition of the following type:

```
comp_t;      /* floating point 13-bit fraction, 3-bit exponent */
```

A structure *acct* with the following members:

```
char    ac_flag;      /* Accounting flag */
char    ac_stat;      /* Exit status */
ushort  ac_uid;       /* Accounting user ID */
ushort  ac_gid;       /* Accounting group ID */
dev_t   ac_tty;       /* Control typewriter */
time_t  ac_btime;     /* Beginning time */
comp_t  ac_utime;     /* Acctng user time in clock ticks */
comp_t  ac_stime;     /* Acctng system time in clock ticks */
comp_t  ac_etime;     /* Acctng elapsed time in clock ticks */
comp_t  ac_mem;       /* Memory usage in clicks */
comp_t  ac_io;        /* Chars trnsfrd by read/write */
comp_t  ac_rw;        /* Number of block reads/writes */
char    ac_comm[8];   /* Command name */
```

A definition of the following symbolic names:

```
AFORK    /* has executed fork, but no exec */
ASU      /* used super-user privileges */
ACCTF    /* record type: 00 = acct */
```

In *ac\_flag*, the *AFORK* flag is turned on by each *fork(2)* and turned off by an *exec(2)*. The *ac\_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds to *ac\_mem* the current process size, computed as follows:

$$\frac{(\text{data size}) + (\text{text size})}{(\text{number of in-store processes using text})}$$

## SEE ALSO

*acct(2)*, *exec(2)*, *exit(2)*, *fork(2)*.

## APPLICATION USAGE

The *ac\_mem* value for a short-lived command gives little information about the actual size of the command, because *ac\_mem* may be incremented while a different command (eg. the shell) is being executed by the process.

## CHANGE HISTORY

## Issue 1

Issue 1 of the SVID, in Appendix K\_EXT 3.0, "Header Files", defines only the typedef, the structure members and the list of symbols. All the other information derives from the equivalent UNIX System V Release 2.0 entry.

## Issue 2

The call for inclusion of `<sys/types.h>` has been added.



## NAME

cpio — format of cpio archive

## DESCRIPTION

When the `—c` option of `cpio(1)` is used, the *header* information is described by:

```
printf or scanf("%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
    &Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
    &Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
    &Longtime, &Hdr.h_namesize,&Longfile,Hdr.h_name);
```

*Longtime* and *Longfile* are equivalent to *Hdr.h\_mtime* and *Hdr.h\_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. *H\_magic* contains the constant 070707 (octal). The meanings of the items *h\_dev* through *h\_mtime* are explained in *stat(2)*. The length of the null-terminated path name *h\_name*, including the null byte, is given by *h\_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h\_filesize* equal to zero.

## SEE ALSO

`cpio(1)`, `stat(2)`, `limits(5)`.

## APPLICATION USAGE

"XVS SOURCE CODE TRANSFER" should be consulted for procedures to ensure portability.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

This description is not a part of the SVID. It is derived from the UNIX System V Release 2.0 definition.



## NAME

group — group file

## DESCRIPTION

*Group* contains for each group the following information:

- group name
- encrypted password
- numerical group ID
- comma-separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line.

This file resides in directory */etc*. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group IDs to names.

## FILES

*/etc/group*

## SEE ALSO

*crypt(3C)*, *passwd(4)*, *grp(5)*.

## CHANGE HISTORY

## Issue 1

This *group* file is not part of Issue 1 of the SVID. It is derived from the UNIX System V release 2.0 definition.

## Issue 2

The sentence "If the password field is null, no password is demanded." has been deleted from the second paragraph.





## NAME

passwd — password file

## DESCRIPTION

The file */etc/passwd* contains, for each user, the following information:

- 1 name
- 2 encrypted password (may be empty)
- 3 numerical user ID
- 4 numerical group ID (may be empty)
- 5 reserved field
- 6 initial working directory
- 7 program to use as shell (may be empty)

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. Fields 2, 4 and 7 may be empty. However if they are not empty, they must be used for their stated purpose. Field 5 is a free field that is implementation specific. Fields beyond 7 are also free but may be standardised in the future. Each user's entry is separated from the next by a new-line.

This file resides in directory */etc*. It has general read permission and can be used, for example, to map numerical user IDs to names.

*Name* is a character string that identifies a user. Its composition should follow the same rules as for file names.

By convention the last element in the pathname of the initial working directory is typically *name*.

## FILES

*/etc/passwd*

## SEE ALSO

*crypt(3C)*, *pwd(5)*.

## CHANGE HISTORY

## Issue 1

Identical to the entry in Appendix 2.9 "Passwd File Format" of Issue 1 of the SVID, except that the SVID does not number the table, and uses the word "null" instead of "empty" when describing passwords in the **DESCRIPTION**.

## Issue 2

The paragraph on password alphabet and ageing has been deleted as it was system-dependent and therefore erroneously included in Issue 1.





## NAME

utmp, wtmp — utmp and wtmp entry formats

## SYNOPSIS

```
#include <sys/types.h>
#include <utmp.h>
```

## DESCRIPTION

These files, which hold user and accounting information, have the structure as defined by `<utmp.h>`. The header file declares the following symbolic names and structure members:

```
UTMP_FILE    /* pathname of utmp file */
WTMP_FILE    /* pathname of wtmp file */
ut_name
```

The structure *utmp* contains the following members:

```
char          ut_user[8];    /* User login name */
char          ut_id[4];      /* /etc/inittab id (usually line #) */
char          ut_line[12];   /* device name (console, lnx) */
short         ut_pid;        /* process id */
short         ut_type;       /* type of entry */
struct exit_status ut_exit;  /* The exit status of a process
                             * marked as DEAD_PROCESS. */
time_t        ut_time;      /* time entry was made */
```

The structure *exit\_status* contains the following members:

```
short  e_termination;
short  e_exit;
```

Definitions for *ut\_type*:

```
EMPTY
RUN_LVL
BOOT_TIME
OLD_TIME
NEW_TIME
INIT_PROCESS    /* Process spawned by "init" */
LOGIN_PROCESS   /* A "getty" process waiting for login */
USER_PROCESS    /* A user process */
DEAD_PROCESS
ACCOUNTING
UTMAXTYPE       /* Largest legal value of ut_type */
```

Special strings or formats used in the *ut\_line* field when accounting for something other than a process. No string for the *ut\_line* field can be more than 11 chars + a NULL character in length.

```
RUNLVL_MSG
BOOT_MSG
OTIME_MSG
NTIME_MSG
```

## SEE ALSO

getut(3C).

## APPLICATION USAGE

Chapter 1, Caveats, warns that the type of *ut\_pid* may change from **short** as declared above. This will not cause problems to most applications provided that they do not use a **short** explicitly to hold the value of *ut\_pid*; it is recommended that only **int** or longer variables are used for this purpose. No type-dependent problems will be caused if the field is accessed by name, as in the following example:

```
#include <stdio.h>
#include <sys/types.h>
#include <utmp.h>
main(){
    int pid;
    struct utmp *getutent(), *utp;

    while((utp = getutent()) != NULL){
        pid = utp->ut_pid;
        printf("Value of pid = %d\n", pid);
        printf("Line %12s\n", utp->ut_line);
    }
}
```

Problems will only occur if the field is accessed by reference (i.e. its address is taken).

## CHANGE HISTORY

## Issue 1

These accounting information files are not described in Issue 1 of the SVID. They are functionally equivalent to the UNIX System V Release 2.0 definition.

## Issue 2

A duplicate **SEE ALSO** section has been removed.

In the second example in the **DESCRIPTION** section, the second occurrence of *ut\_type* has been changed to *ut\_exit*.

## **Header Files**

This chapter describes the contents of header files used for several system calls and library subroutines.

Header files contain the definition of symbolic constants, common structures, preprocessor macros and defined types, typedefs. The library routines in Chapters 2 and 3 specify the header files an application must include in order to use the routine. These files will be present on an applications development system; they do not have to be present on the target execution system.

This Chapter describes the following in each of the header files:

- The symbolic names, data types, data structures and macros defined in the header file that an application should use.
- The definition of the manifest constants.
- The system calls and library routines that may be used by an application and which reference the header file.

The header files `<termio.h>`, `<values.h>` and `<limits.h>` are not needed explicitly to support use of the routines defined in Chapters 2 and 3. They are included because they contain information that an application may need to use.

Relationships to the SVID are given as appropriate in the manual pages.





## NAME

acct — per-process accounting records structures

## SYNOPSIS

```
#include <sys/acct.h>
```

## DESCRIPTION

This file is described in *acct(4)*. Although also appropriate to mention it here, the reader is referred to *acct(4)*; this ensures that there are not two pages containing the same information which the reader would have to check for consistency.

## SEE ALSO

*acct(4)*.

## CHANGE HISTORY

## Issue 1

First published in Issue 1.





**NAME**

assert — verify program assertion

**SYNOPSIS**

```
#include <assert.h>
```

**DESCRIPTION**

This file contains the definitions used by *assert*(3X). A typical example of its contents is:

```
#ifdef NDEBUG
#define assert(EX)
#else
extern void _assert();
#define assert(EX) if (EX) ; else _assert("EX", __FILE__, __LINE__)
#endif
```

**SEE ALSO**

assert(3X).

**CHANGE HISTORY**

**Issue 1**

Issue 1 of the SVID, Appendix BASE: 2.5, "Other Library Routines", refers users to *assert*(LIB) for the contents of this file.



## NAME

ctype — character types

## SYNOPSIS

```
#include <ctype.h>
```

## DESCRIPTION

The following macros are defined in this file:

`isalpha(c)`    `c` is a letter.  
`isupper(c)`    `c` is an upper case letter.  
`islower(c)`    `c` is a lower case letter.  
`isdigit(c)`    `c` is a digit.  
`isxdigit(c)`   `c` is a hexadecimal digit [0-9], [A-F] or [a-f].  
`isalnum(c)`    `c` is an alphanumeric (letter or digit).  
`isspace(c)`    `c` is a space, tab, carriage return, new-line, vertical tab or form-feed.  
`ispunct(c)`    `c` is a punctuation character (neither control nor alphanumeric).  
`isprint(c)`    `c` is a printing character, code 040 (space) through 0176 (tilde).  
`isgraph(c)`    `c` is a printing character, like *isprint* except false for space.  
`isctrl(c)`    `c` is a delete character (0177) or a ordinary control character (code less than 040).  
`isascii(c)`    `c` is an ASCII character, code less than 0200.  
`_toupper(c)`   converts the lower case letter `c` to the corresponding upper case letter.  
`_tolower(c)`   converts the upper case letter `c` to the corresponding lower case letter.  
`toascii(c)`    converts `c` to an ASCII character by masking out high order bits.

## APPLICATION USAGE

These all assume the ASCII character set is in use. `_toupper` and `_tolower` both give incorrect results if their arguments are not of the correct lower or upper case respectively.

## SEE ALSO

`conv(3C)`, `ctype(3C)`.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID in Appendix BASE: 2.6, "Header Files".





## NAME

dirent — format of directory entries

## SYNOPSIS

```
#include <sys/types.h>
#include <limits.h>
#include <sys/dirent.h>
```

## DESCRIPTION

Defines the following data type through typedef:

DIR    A structure containing information about a directory.

The structure *dirent* is defined, containing the following members:

```
ino_t    d_ino;                        /* inode number */
char    d_name[NAME_MAX+1];        /* name of entry */
```

## SEE ALSO

directory(3X).

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

This is not in the SVID. It is specific to X/OPEN systems.





## NAME

PATH, HOME, TERM, TZ, LOGNAME — user environment

## DESCRIPTION

An array of strings called the "environment" is made available by *exec(2)* when a process begins. By convention, these strings have the form "name=value". The following names are used by various commands:

**PATH** The sequence of directory prefixes that some applications apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (:). The logging-in or signing-on procedure will typically set *PATH=:/bin:/usr/bin*.

**HOME** Name of the user's login directory, set from the password file *passwd(4)*.

**TERM** The kind of terminal for which output is to be prepared. This information is used by applications which want to exploit special capabilities of the terminal.

**TZ** Time zone information. The format is *xxxnzzz* where *xxx* is standard local time zone abbreviation, *n* is the difference in hours from GMT, and *zzz* is the abbreviation for the daylight-saving local time zone, if any; for example, *MET-1EET*. See *ctime(3C)*.

**LOGNAME**

The name used during the initial login.

Further names may be placed in the environment by *exec(2)*. It is unwise to conflict with certain variables that are frequently exported by widely used command interpreters: MAIL, PS1, PS2, IFS.

## SEE ALSO

*exec(2)*, *ctime(3C)*, *passwd(4)*.

## FUTURE DIRECTIONS

The number in TZ will be defined as an optional minus sign followed by two hour digits and two minute digits, *[-]hhmm*, in order to represent fractional time zones.

## CHANGE HISTORY

## Issue 1

Issue 1 of the SVID gives the equivalent information in BASE: 2.8 Environmental Variables.

## Issue 2

The format of the description of the number in TZ has been changed to improve clarity.

The description of the LOGNAME variable has been added.



## NAME

errno — system error numbers

## SYNOPSIS

```
#include <errno.h>
```

## DESCRIPTION

The <errno.h> include file contains the following statements:

```
#include <sys/errno.h>
extern int errno;
```

The <sys/errno.h> system include file gives values for the following defined names.

**Note:** The braces around the error names are a typographic convention, and do not form part of the name. The **#define** statements in the header file will look like this:

```
#define E2BIG 12345 /* perhaps.... */
```

and programs using these names will not include the braces.

[E2BIG]	Arg list too long
[EACCES]	Permission denied
[EAGAIN]	Resource unavailable, try again
[EBADF]	Bad file number
[EBUSY]	Device or resource busy
[ECHILD]	No child processes
[EDEADLK]	Deadlock avoided
[EDOM]	Math arg out of domain of func
[EEXIST]	File exists
[EFAULT]	Bad address
[EFBIG]	File too large
[EIDRM]	Identifier removed
[EINTR]	Interrupted system call
[EINVAL]	Invalid argument
[EIO]	I/O error
[EISDIR]	Is a directory
[EMFILE]	Too many open files
[EMLINK]	Too many links
[ENFILE]	File table overflow
[ENODEV]	No such device
[ENOENT]	No such file or directory
[ENOEXEC]	Exec format error
[ENOLCK]	No locks available
[ENOMEM]	Not enough space
[ENOMSG]	No message of desired type
[ENOSPC]	No space left on device
[ENOTBLK]	Block device required
[ENOTDIR]	Not a directory
[ENOTTY]	Not a character device
[ENXIO]	No such device or address
[EPERM]	Not owner
[EPIPE]	Broken pipe
[ERANGE]	Result too large



[EROFS]	Read only file system
[ESPIPE]	Illegal seek
[ESRCH]	No such process
[ETXTBSY]	Text file busy
[EXDEV]	Cross-device link

†The [EFAULT] error is caused by a program referencing data outside its legitimate address space. The reliable detection of this error cannot be guaranteed.

### CHANGE HISTORY

#### Issue 1

Identical to the entry in Issue 1 of the SVID, Appendix BASE 2.6 Header Files, in all the error definitions.

#### Issue 2

Error definitions for interprocess communication added.

## NAME

fcntl — file control options

## SYNOPSIS

```
#include <fcntl.h>
```

## DESCRIPTION

The *fcntl(2)* function provides for control over open files. This include file describes requests and arguments to *fcntl* and *open(2)*.

Flag values accessible to *open(2)* and *fcntl(2)*

(The first three can only be set by *open*)

O\_RDONLY /\* Open for reading only \*/

O\_WRONLY /\* Open for writing only \*/

O\_RDWR /\* Open for reading and writing \*/

O\_NDELAY /\* Non-blocking I/O \*/

O\_APPEND /\* Append (writes guaranteed at the end) \*/

Flag values accessible only to *open(2)*

O\_CREAT /\* Open with file create (uses third open arg) \*/

O\_TRUNC /\* Open with truncation \*/

O\_EXCL /\* Exclusive open \*/

*Fcntl(2)* requests

F\_DUPFD /\* Duplicate fildes \*/

F\_GETFD /\* Get fildes flags \*/

F\_SETFD /\* Set fildes flags \*/

F\_GETFL /\* Get file flags \*/

F\_SETFL /\* Set file flags \*/

F\_GETLK /\* Get blocking file locks \*/

F\_SETLK /\* Set or clear file locks and fail on busy \*/

F\_SETLKW /\* Set or clear file locks and wait on busy \*/

F\_RDLCK /\* Read lock \*/

F\_WRLCK /\* Write lock \*/

F\_UNLCK /\* Remove lock(s) \*/

The structure *flock* contains the following members:

file segment locking control structure

short l\_type; /\* Type of file \*/

short l\_whence; /\* Starting offset \*/

long l\_start; /\* Relative offset \*/

long l\_len; /\* If 0 then until EOF \*/

int l\_pid; /\* Returned with F\_GETLK \*/

## SEE ALSO

fcntl(2), open(2).

## CHANGE HISTORY

## Issue 1

Functionally equivalent to the definition in Issue 1 of the SVID, Appendix BASE: 2.6 Header Files.

The type of *l\_pid* in struct *flock* has been corrected to *int*.

## Issue 2

References to `O_SYNC` and `I_sysid` have been removed.



## NAME

ftw — file tree walk

## SYNOPSIS

```
#include <ftw.h>
```

## DESCRIPTION

Codes for the third argument to the user-supplied function which is passed as the second argument to *ftw*(3C):

```
FTW_F      /* File */  
FTW_D      /* Directory */  
FTW_DNR    /* Directory without read permission */  
FTW_NS     /* Unknown type, stat failed */
```

## SEE ALSO

*ftw*(3C).

## CHANGE HISTORY

## Issue 1

Functionally equivalent to the entry in Issue 1 of the SVID Appendix BASE: 2.6 Header Files.



NAME

grp — group structure

SYNOPSIS

```
#include <grp.h>
```

DESCRIPTION

*Struct group* contains the following members:

```
char    *gr_name;  
char    *gr_passwd;  
int     gr_gid;  
char    **gr_mem;
```

SEE ALSO

getgrent(3C).

CHANGE HISTORY

Issue 1

Not in Issue 1 of the SVID.

Issue 2

The reference to *getgrent* has been corrected to section 3C.





## NAME

limits — implementation specific constants

## SYNOPSIS

```
#include <limits.h>
```

## DESCRIPTION

The following names are defined in `<limits.h>` and are used throughout the descriptive text of the X/OPEN System V specification. The values given in the column headed "Portability Value" are the values that applications should assume for portability across all X/OPEN systems. For example, `LOCK_MAX` has a portability value of 32, which means that all X/OPEN systems will be capable of supporting up to 32 entries in their system lock tables. A particular system may be capable of supporting more than 32 entries, in which case its `<limits.h>` file will set `LOCK_MAX` to a higher value, but any application assuming this higher number is not guaranteed to be portable to all systems.

The items at the end of the list ending in "`_MIN`" give the most negative values that the mathematical types are guaranteed to be capable of representing. Numbers of a more negative value may be supported on some systems, as indicated by their `<limits.h>` file, but applications requiring such numbers are not guaranteed to be portable to all systems.

The symbol `*` in the Portability Value column indicates that there is no guaranteed value across all X/OPEN systems.

By inspecting the `limits.h` file on a specific system an applications writer can determine the actual limits in operation. Similarly, by *including* the file in the compilation an application can test the appropriate limits to determine whether it can operate on a particular system, or it may even alter its behaviour to match the system thus making itself portable across a varying range of limit settings/systems.

Name	Description	Portability Value
<code>ARG_MAX</code>	Max length of argument to <code>exec(2)</code> including environment data	4096
<code>CHAR_BIT</code>	Number of bits in a <code>char</code>	8
<code>CHAR_MAX</code>	Max integer value of a <code>char</code>	127
<code>CHILD_MAX</code>	Max number of processes per user ID	4
<code>CLK_TCK</code>	Number of clock ticks per second	10
<code>DBL_DIG</code>	Digits of precision of a <code>double</code>	*
<code>FCHR_MAX</code>	Max size of a file in bytes	1000000
<code>FLT_DIG</code>	Digits of precision of a <code>float</code>	*
<code>FLT_MAX</code>	Max decimal value of a <code>float</code>	*
<code>INT_MAX</code>	Max decimal value of an <code>int</code>	32767
<code>LINK_MAX</code>	Max number of links to a single file	8
<code>LOCK_MAX</code>	Max number of entries in system lock table	32
<code>LONG_BIT</code>	Number of bits in a <code>long</code>	32
<code>LONG_MAX</code>	Max decimal value of a <code>long</code>	2147483647
<code>DBL_MAX</code>	Max decimal value of a <code>double</code>	*
<code>MAX_CHAR</code>	Max size of character input buffer	*
<code>NAME_MAX</code>	Max number of characters in a file name	14
<code>OPEN_MAX</code>	Max number of files a process can have open	16
<code>PASS_MAX</code>	Max number of significant characters in a password	8

## LIMITS(5)

Header Files

PATH_MAX	Max number of characters in a path name	255
PID_MAX	Max value for a process ID	32000
PIPE_BUF	Max number bytes atomic in write to a pipe	512
PIPE_MAX	Max number bytes written to a pipe in a write	4096
PROC_MAX	Max number of simultaneous processes, system wide	8
SHRT_MAX	Max decimal value of a <b>short</b>	32767
SYSPID_MAX	Max pid of system processes	1
STD_BLK	Number bytes in a physical I/O block	256
SYS_NMLN	Number of chars in uname-returned strings	8
SYS_OPEN	Max number of files open on system	16
TMP_MAX	Max number of unique names generated by <i>tmpnam</i> (3S)	10000
UID_MAX	Max value for a user or group ID	32000
USI_MAX	Max decimal value of an <b>unsigned</b>	65535
WORD_BIT	Number of bits in a "word" or int	16
CHAR_MIN	Min integer value of a <b>char</b>	0
DBL_MIN	Min decimal value of a <b>double</b>	*
FLT_MIN	Min decimal value of a <b>float</b>	*
INT_MIN	Min decimal value of a <b>int</b>	-32768
LONG_MIN	Min decimal value of a <b>long</b>	-2147483648
SHRT_MIN	Min decimal value of a <b>short</b>	-32768

### SEE ALSO

Chapter 1 (Caveats: limits.h and values.h), values(5).

### CHANGE HISTORY

#### Issue 1

The SVID does not define a `<limits.h>` include file and therefore does not allow for applications to determine the settings on a given system. It does use the listed names in place of absolute values in the descriptive text.

#### Issue 2

The words "including environment data" have been included in the description of ARG\_MAX.



NAME

lock — process lock types

SYNOPSIS

```
#include <sys/lock.h>
```

DESCRIPTION

Values are given for the following flags for locking processes and texts:

PROLOCK	lock text and data segments into memory (process lock)
TXTLOCK	lock text segment into memory (text lock)
DATLOCK	lock data segment into memory (data lock)
UNLOCK	remove locks

SEE ALSO

plock(2).

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID. This include file is identified in Appendix K\_EXT: 3.0 and its contents are given in *plock*(KEXT).



## NAME

malloc — main memory allocator

## SYNOPSIS

```
#include <malloc.h>
```

## DESCRIPTION

This file provides definitions for constants defining *mallopt*, see *malloc(3X)* operations and declares the structure *mallinfo* to contain the members shown.

```
M_MXFAST    /* set size of blocks to be fast */
M_NLBLKS    /* set number of block in a holding block */
M_GRAIN      /* set number of sizes mapped to one, for
              small blocks */
M_KEEP       /* retain contents of block after a free until
              another allocation */
```

The structure *mallinfo* contains:

```
int arena;    /* total space in arena */
int ordblks;  /* number of ordinary blocks */
int smblks;   /* number of small blocks */
int hblks;    /* number of holding blocks */
int hblkhd;   /* space in holding block headers */
int usmbks;   /* space in small blocks in use */
int fsmblks;  /* space in free small blocks */
int uordblks; /* space in ordinary blocks in use */
int fordblks; /* space in free ordinary blocks */
int keepcost; /* cost of enabling keep option */
```

## SEE ALSO

*malloc(3X)*.

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID.

This is identical to the definition of *malloc.h* given in the SVID, Appendix BASE: 2.6 Header Files.





## NAME

math — mathematical types

## SYNOPSIS

```
#include <math.h>
```

## DESCRIPTION

Values are given for the following useful constants:

M_E	Value of $e$
M_LOG2E	Value of base 2 $\log e$
M_LOG10E	Value of base 10 $\log e$
M_LN2	Value of base $e \log 2$
M_LN10	Value of base $e \log 10$
M_PI	Value of $\pi$
M_PI_2	Value of $\pi/2$
M_PI_4	Value of $\pi/4$
M_1_PI	Value of $1/\pi$
M_2_PI	Value of $2/\pi$
M_2_SQRTPI	Value of $2/\sqrt{\pi}$
M_SQRT2	Value of $\sqrt{2}$
M_SQRT1_2	Value of $1/\sqrt{2}$

The include file contains a define statement for the MAXFLOAT symbol which is machine dependent, and the value HUGE which is returned for error conditions found in the math library, see **FUTURE DIRECTIONS**, below.

MAXFLOAT	Value of maximum floating point number
HUGE	Error value returned by the math library

The structure *exception* is defined, containing the following members:

```
int type;          /* type of error that occurred */
char *name;        /* name of the function that incurred the error */
double arg1;       /* argument 1 of the invoked function */
double arg2;       /* argument 2 of the invoked function */
double retval;     /* set of default values returned by the function */
```

## FUTURE DIRECTIONS

A macro HUGE\_VAL will be defined to represent error values returned by the math functions. This macro will call a function which will either return  $+\infty$  on a system supporting IEEE P754 standard or  $\{MAXDOUBLE\}$  on a system that does not support the IEEE P754 standard. The functions which currently return HUGE or  $\pm$ HUGE\_VAL on overflow will return HUGE\_VAL or  $\pm$ HUGE\_VAL respectively.

## SEE ALSO

erf(3M), exp(3M), floor(3M), gamma(3M), hypot(3M), sinh(3M), trig(3M).

## CHANGE HISTORY

## Issue 1

Functionally equivalent to Issue 1 of the SVID in Appendix BASE: 2.6, Header Files.

## Issue 2

The mathematical symbols  $\pi$  and  $\sqrt{\phantom{x}}$  have replaced their textual equivalents.





**NAME**

memory — memory operations

**SYNOPSIS**

```
#include <memory.h>
```

**DESCRIPTION**

This file declares the types of the functions performing memory operations.

**SEE ALSO**

memory(3C).

**FUTURE DIRECTION**

The declarations in <memory.h> will be moved to <string.h>.

**CHANGE HISTORY**

**Issue 1**

Functionally equivalent to Issue 1 of the SVID in Appendix BASE: 2.6, Header Files.



## NAME

mon — prepare execution profile

## SYNOPSIS

```
#include <mon.h>
```

## DESCRIPTION

The following structures are declared, and their members indicated:

```
struct hdr
char    *lpc;    /* low PC of range being profiled */
char    *hpc;    /* high PC of range being profiled */
int      nfns;   /* number of cnt structures */
```

```
struct cnt
char    *fnpc;   /* function PC */
long    mcnt;    /* call count */
```

A **typedef** is given for type *WORD*.

Definitions are given for the following names:

```
MON_OUT    filename for profile, e.g. "mon.out"
MPROGS0
MSCALE0
NULL       zero
```

## SEE ALSO

monitor(3C).

## CHANGE HISTORY

## Issue 1

This include file is not defined in the SVID.





## NAME

pwd — password file structure

## SYNOPSIS

```
#include <pwd.h>
```

## DESCRIPTION

Definitions are given for the following structures and their members.

```
struct passwd
```

```
char *pw_name;      /* name */
char *pw_passwd;    /* encrypted password (may be empty) */
int pw_uid;         /* numerical user ID */
int pw_gid;         /* numerical group ID (may be empty) */
char *pw_age;       /* password age */
char *pw_comment;
char *pw_gecos;      /* free field */
char *pw_dir;        /* initial working directory */
char *pw_shell;      /* program to use as shell (may be empty) */
```

```
struct comment
```

```
char *c_dept;
char *c_name;
char *c_acct;
char *c_bin;
```

## SEE ALSO

getpwent(3C), putpwent(3C).

## CHANGE HISTORY

## Issue 1

This include file is not defined in the SVID.





NAME

search — hash search tables

SYNOPSIS

```
#include <search.h>
```

DESCRIPTION

Defines ENTRY as the structure *entry* through a typedef. *Entry* includes the following members:

```
char *key, *data;
```

Defines ACTION and VISIT as enumeration data types through typedefs as follows:

```
enum { FIND, ENTER } ACTION;
```

```
enum { preorder, postorder, endorder, leaf } VISIT;
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C), tsearch(3C).

CHANGE HISTORY

Issue 1

Identical to Issue 1 of the SVID in Appendix BASE: 2.6 Header Files.



NAME

setjmp — stack environment type

SYNOPSIS

#include <setjmp.h>

DESCRIPTION

The include file contains a statement defining the symbolic constant \_JBLEN which is machine dependent. A typedef is provided for type *jmp\_buf*.

SEE ALSO

setjmp(3C).

CHANGE HISTORY

Issue 1

The SVID does not define this include file.





## NAME

signal — signals

## SYNOPSIS

```
#include <signal.h>
```

## DESCRIPTION

The `<signal.h>` include file contains definitions of the following symbolic names:

SIGABRT	Process abort signal.
SIGALRM	Alarm clock.
SIGFPE	Floating point exception.
SIGHUP	Hangup.
SIGILL	Illegal instruction (not reset when caught).
SIGINT	Interrupt.
SIGKILL	Kill (cannot be caught or ignored).
SIGPIPE	Write on a pipe with no one to read it.
SIGQUIT	Quit.
SIGSEGV	Segmentation violation.
SIGSYS	Bad argument to system call.
SIGTERM	Software termination signal from kill.
SIGTRAP	Trace trap (not reset when caught).
SIGUSR1	User defined signal 1.
SIGUSR2	User defined signal 2.

The include file contains a statement to define the symbolic constant `NSIG` which is machine dependent, and also the names `SIG_DFL` and `SIG_IGN`.

## FUTURE DIRECTIONS

A macro `SIG_ERR` will be defined in `<signal.h>` to represent the return value `(int(*)())-1` by `signal(2)` in case of error.

## APPLICATIONS USAGE

Refer back to Chapter 1 on portable signals.

## CHANGE HISTORY

## Issue 1

This is identical to the Issue 1 of the SVID definition in Appendix BASE: 2.6 Header Files, except that `SIGSEGV` has been added from System V, Release 2.0 and `SIGABRT` is a future direction in the SVID.





## NAME

stat — data returned by stat system call

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
```

## DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file.

The structure *stat* contains the following members:

```
dev_t    st_dev;      /* device with directory entry for
                      file */
ino_t    st_ino;      /* inode number */
ushort   st_mode;     /* type of file (see below) */
short    st_nlink;    /* number of links */
ushort   st_uid;      /* user ID of file owner */
ushort   st_gid;      /* group ID of file owner */
dev_t    st_rdev;     /* device (if file is character or
                      block special) */
off_t    st_size;     /* file size in bytes */
time_t   st_atime;    /* time of last access */
time_t   st_mtime;    /* time of last data modification */
time_t   st_ctime;    /* time of last status change */
```

Symbolic names for the values of *st\_mode* as well as macros to manipulate this field are also defined:

File type:

```
S_IFMT   /* type of file */
S_IFBLK  /* block special */
S_IFCHR  /* character special */
S_IFDIR  /* directory */
S_IFIFO  /* fifo special */
S_IFREG  /* regular */
```

File modes:

```
S_ISUID  /* set user ID on execution */
S_ISGID  /* set group ID on execution */
S_ISVTX+ /* SEE APPLICATION USAGE BELOW */
S_IRWXU  /* read, write, execute/search by owner */
S_IRUSR  /* read permission, owner */
S_IWUSR  /* write permission, owner */
S_IXUSR  /* execute/search permission, owner */
S_IRWXG  /* read, write, execute/search by group */
S_IRGRP  /* read permission, group */
S_IWGRP  /* write permission, group */
S_IXGRP  /* execute/search permission, group */
S_IRWXO  /* read, write, execute/search by others */
S_IROTH  /* read permission, others */
S_IWOTH  /* write permission, others */
S_IXOTH  /* execute/search permission, others */
```

```
S_IREAD    /* read premission, owner */
S_IWRITE    /* write premission, owner */
S_IEXEC     /* execute/search premission, owner */
```

The following macros are defined:

```
S_ISBLK()   /* test for a block special file */
S_ISCHR()   /* test for a character special file */
S_ISDIR()   /* test for a directory */
S_ISFIFO()  /* test for a fifo special file */
S_ISREG()   /* test for a regular file */
```

#### SEE ALSO

mknod(2), stat(2), types(5).

#### APPLICATIONS USAGE

Use of the macros is recommended for determining the type of a file.

It should be noted that S\_IREAD, S\_IWRITE and S\_IEXEC are duplicated with the names S\_IRUSR, S\_IWUSR and S\_IXUSR and the latter are preferred for portability as they are compatible with the /usr/group standard.

S\_ENFMT: record locking enforcement is not currently part of the XVS but this name is reserved for future use.

†S\_ISVTX: Although this name is defined, the "save swapped text after use" functionality is becoming redundant and its use is not recommended.

#### CHANGE HISTORY

##### Issue 1

The *stat* structure is identical to that given in Issue 1 of the SVID, Appendix BASE 2.6: Header Files. The remainder is taken from the SVID **FUTURE DIRECTION** statement in Appendix BASE 1.6: Comparison to 1984 /usr/group Standard.

## NAME

stdio — standard buffered input/output

## SYNOPSIS

```
#include <stdio.h>
```

## DESCRIPTION

Defines the following symbolic names:

```
BUFSIZ    /* Size of stdio buffers */
NULL      /* NULL stdio pointer */
EOF        /* End of File return value */
stdin     /* File pointer to standard input */
stdout    /* File pointer to standard output */
stderr    /* File pointer to standard error output */
```

Defines the following data type through typedef:

```
FILE      A structure containing information about a file.
```

## APPLICATIONS USAGE

The following names may also be defined in this header file, and should only be used by applications developers in accordance with the definitions (where given) in other interface specifications.

```
L_ctermid  L_cuserid  L_tmpnam
P_tmpdir  _IOERR      _IOFBF
_IOLBF     _IOMYBUF   _IONBF
_IOREAD    _IORW      _IOWRT
_NFILE     _SBFSIZE   _bufend
_bufsiz    clearerr   feof
ferror     fileno     getc
getchar    putc       putchar
```

## SEE ALSO

bsearch(3C), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), hsearch(3C), lsearch(3C), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), stdio(3S), system(3S), tmpfile(3S), tmpnam(3S), tsearch(3C), ungetc(3S), vprintf(3S).

## CHANGE HISTORY

## Issue 1

Identical to Issue 1 of the SVID, Appendix BASE 2.6: Header Files.

## Issue 2

An erroneous reference to \_SBSIZE has been corrected to \_SBFSIZE.





**NAME**

string — string operations

**SYNOPSIS**

**#include** <string.h>

**DESCRIPTION**

The types of the string functions are declared.

**SEE ALSO**

string(3C).

**FUTURE DIRECTIONS**

The SVID mentions that declarations in <memory.h> will be moved to <string.h>.

**CHANGE HISTORY**

**Issue 1**

Derived from the entry in Issue 1 of the SVID in Appendix BASE 2.6: Header Files.





## NAME

termio — define values for termio and ioctl

## SYNOPSIS

```
#include <termio.h>
```

## DESCRIPTION

This file contains the definitions used by *termio*(7) and *ioctl*(2). Refer to those descriptions for the structures and names defined.

## SEE ALSO

ioctl(2), termio(7).

## CHANGE HISTORY

## Issue 1

The SVID entry in Appendix BASE: 2.6, Header Files, likewise refers users to *ioctl*(OS) and *termio*(DEV) for the contents of this file.



## NAME

time — time types

## SYNOPSIS

```
#include <time.h>
```

## DESCRIPTION

The structure *tm* is declared, containing the following members:

```
int    tm_sec;  
int    tm_min;  
int    tm_hour;  
int    tm_mday;  
int    tm_mon;  
int    tm_year;  
int    tm_wday;  
int    tm_yday;  
int    tm_isdst;
```

## SEE ALSO

ctime(3C).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID in Appendix BASE: 2.6 Header Files.





## NAME

times — file access and modification times structure

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/times.h>
```

## DESCRIPTION

The structure returned by *times(2)*, *struct tms*, contains the following members:

time_t tms_utime;	/* User time */
time_t tms_stime;	/* System time */
time_t tms_cutime;	/* User time, children */
time_t tms_cstime;	/* System time, children */

## SEE ALSO

times(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The SVID defines *<sys/times.h>* only within the *times* (OS) description. It is not mentioned in Appendix BASE: 2.6 Header Files.





## NAME

types — primitive system data types

## SYNOPSIS

```
#include <sys/types.h>
```

## DESCRIPTION

Some data types used in system code are implementation dependent. These are defined in the `<types.h>` include file, which contains definitions for at least the following types:

<code>daddr_t</code>	Used for disk block addresses
<code>ushort</code>	Unsigned short
<code>ino_t</code>	Used for file serial numbers
<code>time_t</code>	Used for system time
<code>dev_t</code>	Used for device numbers
<code>off_t</code>	Used for file sizes
<code>key_t</code>	Used for inter-process communication

Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file.

## SEE ALSO

`ctime(3C)`, `stat(2)`, `times(2)`, `utime(2)`.

## CHANGE HISTORY

## Issue 1

The type `daddr_t` is not defined in Issue 1 of the SVID, the other types are defined in Appendix BASE: 2.6 Header Files.

## Issue 2

The type `key_t` has been added because the IPC routines have been added as an optional facility.



## NAME

unistd — standard symbolic constants and structures

## SYNOPSIS

```
#include <unistd.h>
```

## DESCRIPTION

The file defines the symbolic constants and structures which are referenced elsewhere in the standard and which are not already defined or declared in some other "include" file. The contents of this file is shown below:

Symbolic constants for the *access(2)* function:

```
R_OK    /* Test for "Read" Permission */
W_OK    /* Test for "Write" Permission */
X_OK    /* Test for "Execute"(Search) Permission */
F_OK    /* Test for existence of file */
```

Symbolic constants for the *lockf(3C)* function:

```
F_ULOCK /* Unlock a previously locked region */
F_LOCK  /* Lock a region for exclusive use */
F_TLOCK /* Test and lock a region for exclusive use */
F_TEST  /* Test a region for a previous lock */
```

Symbolic constants for the *lseek(2)* function:

```
SEEK_SET /* Set file pointer to "offset" */
SEEK_CUR /* Set file pointer to current plus "offset" */
SEEK_END /* Set file pointer to EOF plus "offset" */
```

Path names of the *passwd* and *group* files:

```
GF_PATH /* Path name of the "group" file */
PF_PATH /* Path name of the "passwd" file */
IF_PATH /* Path name for <...> files */
```

## SEE ALSO

fcntl(2), open(2).

## CHANGE HISTORY

## Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The SVID defines only *F\_ULOCK*, *F\_LOCK*, *F\_TLOCK* and *F\_TEST*. These are defined within the description of *lockf(3C)* and not in Appendix BASE: 2.6; the rest is described in Appendix BASE: 1.6, as a future direction.





## NAME

ustat — file system statistics

## SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>
```

## DESCRIPTION

*Struct ustat* declares at least the following members:

daddr_t	f_tfree;	/* Total free */
ino_t	f_tinode;	/* Total inodes free */
char	f_fname[6];	/* Filsys name */
char	f_fpack[6];	/* Filsys pack name */

## SEE ALSO

ustat(2).

## CHANGE HISTORY

## Issue 1

Identical to the entry in Issue 1 of the SVID, Appendix BASE: 2.6 Header Files.

## Issue 2

The call for inclusion of `<sys/types.h>` has been added.





**NAME**

utmp — utmp file structure

**SYNOPSIS**

Refer to *utmp*(4).

**SEE ALSO**

*utmp*(4).

**CHANGE HISTORY**

**Issue 1**

First published in Issue 1.



## NAME

utsname — system name structure

## SYNOPSIS

```
#include <limits.h>
#include <sys/utsname.h>
```

## DESCRIPTION

This file declares *struct utsname* which includes the following members:

```
char    sysname[SYS_NMLN];
char    nodename[SYS_NMLN];
char    release[SYS_NMLN];
char    version[SYS_NMLN];
char    machine[SYS_NMLN];
```

## SEE ALSO

uname(2).

## CHANGE HISTORY

### Issue 1

Identical to Issue 1 of the SVID, Appendix BASE: 2.6 Header Files.

### Issue 2

A call for the inclusion of `<limits.h>` has been added as it was incorrectly omitted in Issue 1.





## NAME

values — machine-dependent values

## SYNOPSIS

```
#include <values.h>
```

## DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures.

The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS PER BYTE	The number of bits in a byte.
BITS( <i>type</i> )	The number of bits in a specified <i>type</i> (e.g., int).
HIBITS	The value of a short integer with only the high-order bit set (in most implementations, 0x8000).
HIBITL	The value of a long integer with only the high-order bit set (in most implementations, 0x80000000).
HIBITI	The value of a regular integer with only the high-order bit set (usually the same as HIBITS or HIBITL).
MAXSHORT	The maximum value of a signed short integer (in most implementations, 0x7FFF $\equiv$ 32767).
MAXLONG	The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF $\equiv$ 2147483647).
MAXINT	The maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG).
DMAXEXP	The maximum exponent of a <b>double</b> .
FMAXEXP	The maximum exponent of a <b>float</b> .
DMINEXP	The minimum exponent of a <b>double</b> .
FMINEXP	The minimum exponent of a <b>float</b> .
DMAXPOW2	The largest power of two exactly representable as a <b>double</b> .
FMAXPOW2	The largest power of two exactly representable as a <b>float</b> .
_FEXPLEN	The number of bits for the exponent of a <b>float</b> .
_EXPBASE	The exponent base.
_DEXPLEN	The number of bits for the exponent of a <b>double</b> .
_IEEE	1 if the IEEE standard representation is used.
_LENBASE	Number of bits in the exponent base.
_HIDDENBIT	1 if high-significance bit in the mantissa is implicit. The largest power of two exactly representable as a <b>double</b> .
FSIGNIF	The number of significant bits in the mantissa of a single-precision floating-point number.
DSIGNIF	The number of significant bits in the mantissa of a double-precision floating-point number.

MAXFLOAT, LN_MAXFLOAT	The maximum value of a single-precision floating-point number, and its natural logarithm.
MAXDOUBLE, LN_MAXDOUBLE	The maximum value of a double-precision floating-point number, and its natural logarithm.
MINFLOAT, LN_MINFLOAT	The minimum positive value of a single-precision floating-point number, and its natural logarithm.
MINDOUBLE, LN_MINDOUBLE	The minimum positive value of a double-precision floating-point number, and its natural logarithm.

### SEE ALSO

limits(5), math(5).

### CHANGE HISTORY

#### Issue 1

Identical to Issue 1 of the SVID, Appendix BASE: 2.6 Header Files, except that the names LN\_MINFLOAT and LN\_MAXFLOAT have been included from System V Release 2.0.

#### Issue 2

The manifest constant HIDDENBIT was corrected to \_HIDDENBIT.

The FILES section, which was erroneously included in Issue 1, was deleted from Issue 2.



## NAME

varargs — handle variable argument list

## SYNOPSIS

```
#include <varargs.h>

va_alist
va_dcl

void va_start(pvar)
va_list pvar;

type va_arg(pvar, type)
va_list pvar;

void va_end(pvar)
va_list pvar;
```

## DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists (such as *printf(3S)*) but do not use *varargs* are inherently nonportable, as different machines use different argument-passing conventions.

*va\_alist* is used as the parameter list in a function header.

*va\_dcl* is a declaration for *va\_alist*. No semicolon should follow *va\_dcl*.

*va\_list* is a type defined for the variable used to traverse the list.

*va\_start* is called to initialise *pvar* to the beginning of the list.

*va\_arg* will return the next argument in the list pointed to by *pvar*. *Type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

*va\_end* is used to clean up.

Multiple traversals, each bracketed by *va\_start* ... *va\_end*, are possible.

## EXAMPLE

This example is a possible implementation of *execl(2)*.

```
#include <varargs.h>
#define MAXARGS 100

/*      execl is called by
        execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
{
    va_list ap;
    char *file;
    char *args[MAXARGS];
    int argno = 0;

    va_start(ap);
```

```
    file = va_arg(ap, char *);
    while ((args[argno+ +] =
        va_arg(ap, char *)) != (char *)0)
        ;
    va_end(ap);
    return execv(file, args);
}
```

## SEE ALSO

exec(2), printf(3S).

## APPLICATION USAGE

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame. For example, *exec1* is passed a zero pointer to signal the end of the list. *Printf* can tell how many arguments are there by the format.

It is non-portable to specify a second argument of **char**, **short**, or **float** to *va\_arg*, since arguments seen by the called function are not **char**, **short**, or **float**. C converts **char** and **short** arguments to **int** and converts **float** arguments to **double** before passing them to a function.

## CHANGE HISTORY

### Issue 1

Derived from the entry in Issue 1 of the SVID with the following change:

The SVID simply mentions the declaration, typedef or definition of the names *va\_list*, *va\_start*, *va\_end*, *va\_arg* and *va\_decl* in Appendix BASE: 2.6 Header Files. The application writer is given no hint how to use them.



## **Chapter 6**

This chapter is reserved for future use.





## **Special Files**

This chapter describes various special files which are present in all X/OPEN systems. Most systems will contain other entries for specific devices.

The entries shown here are mandatory on all systems, except for the source code transfer devices *sct(7)* which are only mandatory on systems with the relevant hardware.

Where there are corresponding entries in the SVID, they are to be found in Appendix 2.11, "Special Device Files".





NAME

console — system console interface

DESCRIPTION

**/dev/console** is a generic name given to the system console. It is usually linked to a particular machine dependent special file. It provides a basic I/O interface to the system console.

FILES

/dev/console

SEE ALSO

termio(7).

CHANGE HISTORY

Issue 1

Identical to the definition in Issue 1 of the SVID, Appendix 2.11, "Special Device Files", except that the SVID states that the system console works through the *termio* interface. This is not necessarily true of X/OPEN systems.



**NAME**

null — the null file

**DESCRIPTION**

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

**FILES**

/dev/null

**CHANGE HISTORY**

**Issue 1**

Derived from the entry in Issue 1 of the SVID in Appendix 2.11, "Special Device Files", except that the SVID also states that the "output of a command is written to the special file */dev/null* when the command is executed for its side effects and not for its output". This information is not relevant to applications development.





**NAME**

sctmt, sctfd — source code transfer devices (**OPTIONAL**)

**DESCRIPTION**

The files `/dev/sctmt{lmh}` and `/dev/sctfd{lm}` are the names of the special files used to transfer software between X/OPEN systems. Their descriptions follow.

**½ Inch Magnetic Tape**

The standard physical tape recording format is:

- 9 track Phase Encoded (PE), 1600 bits per inch (bpi).

Optional formats that may also be supported by particular systems in addition to this are:

- 9 track Group Code Recording (GCR), 6250 bpi.
- 9 track Non Return to Zero Inverted (NRZI), 800 bpi.

**Special File Names and Blocking**

The device names associated with these formats are as follows:

name	format	blocksize(bytes)
<code>/dev/sctmtl&lt;number&gt;</code>	NRZI	512
<code>/dev/sctmtm&lt;number&gt;</code>	PE	512
<code>/dev/sctmth&lt;number&gt;</code>	GCR	512
<code>/dev/rsctmtl&lt;number&gt;</code>	NRZI	see below
<code>/dev/rsctmtm&lt;number&gt;</code>	PE	see below
<code>/dev/rsctmth&lt;number&gt;</code>	GCR	see below

Note that on the "raw" devices (rsct...), data is both read and written in blocks corresponding to the length requested in the read or write system call (see `read(2)` and `write(2)`).

The device names are usually links to system-specific device names.

**Device Numbers**

The part of the special file name described in the table above as `<number>` is constructed from the system specific unit number with the addition of 0 or 128 (decimal) to indicate whether the tape is to be rewound on closure. Any tape that is opened for writing has a tape mark written on closure. Addition of 0 to the unit number causes the tape to be rewound to the beginning of tape mark (BOT); addition of 128 inhibits this.

Hosted implementations may need extra information to be specified in the device names, for example volume names.

**5 ¼ Inch Floppy Disk Exchange****Physical Recording**

The standard floppy disc recording formats are as follows:

Floppy Disc Recording Formats	
a)	80 tracks (96 tracks per inch) on each side 2 tracks per cylinder 9 sectors per track 512 bytes per sector Double Sided - Double Density Modified Frequency Modulation (MFM) recording on all the tracks Conformity with standard ECMA-78, track format 2
b)	40 tracks (48 tracks per inch) on each side 2 tracks per cylinder 9 sectors per track 512 bytes per sector Modified Frequency Modulation (MFM) recording on all the tracks Double Sided - Double Density Conformity with standard ECMA-70, track format 2

**Note:** 80 track is the preferred format; systems equipped only with 80 track drives may be able to read but not write 40 track disks.

### Special File Names

The special file names associated with these formats are as follows:

Name	Number of Tracks
/dev/sctfdl<number>	40
/dev/sctfdm<number>	80

The device names are usually links to system specific device names.

### Device Number

The part of the special file name described in the table above as <number> is constructed from a system specific unit number. To this is added 0 or 128 (decimal) to define whether cylinder 0 is accessible. 0 means that cylinder 0 is accessible, so the first sector accessed is sector 1 track 0 cylinder 0. 128 means that cylinder 0 is *not* accessible, so the first sector accessed is sector 1 track 0 cylinder 1.

To conform with ECMA 107 standards, cylinder 0 should only be used for administrative information and the source code files should start in sector 1, track 0, cylinder 1.

### FILES

/dev/sctmtm  
/dev/sctfdm

### SEE ALSO

"XVS SOURCE CODE TRANSFER".

### CHANGE HISTORY

#### Issue 1

This is not in the SVID. It is specific to X/OPEN systems.

#### Issue 2

The description of the floppy disc formats now includes the specification of double sided — double density and conformity with the ECMA standards.



## NAME

termio — general terminal interface

## SYNOPSIS

```
#include <termio.h>
```

## DESCRIPTION

This is the System V interface for locally connected asynchronous lines. Refer to Chapter 1 for a discussion of the caveats with respect to this interface. The description below is relevant only to those lines configured to use this interface.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by the system and become a user's *standard input*, *standard output* and *standard error*. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork(2)*. A process can break this association by changing its process group using *setpgrp(2)*.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, or when an input line exceeds the maximum allowable number of input characters. Currently, this limit is {MAX\_CHAR} characters. When the input limit is reached, all the saved characters may be thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, *erase* and *kill* processing is normally done. The erase character erases the last character typed, except that it will not erase beyond the beginning of the line. The kill character kills (deletes) the entire input line, and optionally outputs a new-line character. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character "\". In this case the escape character is not read. The erase and kill characters may be changed.

## Special Characters

Certain characters have special functions on input. These functions are summarised as follows. Where they cannot be changed, the corresponding characters are indicated in parentheses:

INTR Generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location, see *signal(2)*.

- QUIT Generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but the abnormal termination routines will be executed.
- ERASE Erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.
- KILL Deletes the entire line, as delimited by a NL, EOF, or EOL character.
- EOF May be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.
- NL (ASCII NL) is the normal line delimiter. It can not be changed or escaped.
- EOL Is an additional line delimiter, like NL. It is not normally used.
- STOP (control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.
- START (control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The START/STOP characters can not be changed or escaped.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be redefined by the user. The ERASE, KILL, and EOF characters may be escaped by a preceding “\” character, in which case no special function is done.

When the carrier signal from the data-set drops, a *hang-up* signal, SIGHUP is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hang-up signal is ignored, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

## ioctl(2) Requests

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the *termio* structure, defined in `<termio.h>`:

A definition is given for:

```
NCC    /* size of the array
        * c_cc for special control characters */
```



The structure *termio* includes the following members:

```

unsigned    short    c_iflag; /* input modes */
unsigned    short    c_oflag; /* output modes */
unsigned    short    c_cflag; /* control modes */
unsigned    short    c_lflag; /* local modes */
char        c_line; /* line discipline */
unsigned    char      c_cc[NCC]; /* control chars */

```

The special control characters are defined by the array *c\_cc*. The relative positions for each function are as follows:

```

0  VINTR
1  VQUIT
2  VERASE
3  VKILL
4  VEOF
4  VMIN
5  VEOL
5  VTIME
6  reserved
7  SWTCH

```

Their default values are implementation dependent.

### Input Modes

The *c\_iflag* field describes the basic terminal input control:

```

IGNBRK      Ignore break condition.
BRKINT      Signal interrupt on break.
IGNPAR      Ignore characters with parity errors.
PARMRK      Mark parity errors.
INPCK       Enable input parity check.
ISTRIP      Strip character.
INLCR       Map NL to CR on input.
IGNCR       Ignore CR.
ICRNL       Map CR to NL on input.
IUCLC       Map upper case to lower case on input.
IXON        Enable start/stop output control.
IXANY       Enable any character to restart output.
IXOFF       Enable start/stop input control.

```

If *IGNBRK* is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if *BRKINT* is set, the break condition will generate an interrupt signal and flush both the input and output queues. If *IGNPAR* is set, characters with other framing and parity errors are ignored.

If *PARMRK* is set, a character with a framing or parity error which is not ignored is read as the three-character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if *ISTRIP* is not set, a valid character of 0377 is read as 0377, 0377. If *PARMRK* is not set, a framing or parity error which is not ignored is read as the character NUL (0).



If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper case alphabetic character is translated into the corresponding lower case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read. If IXANY is set, any input character, will restart output which has been suspended.

If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all-bits-clear.

## Output Modes

The `c_oflag` field specifies the system treatment of output:

OPOST	Postprocess output.
OLCUC	Map lower case to upper on output.
ONLCR	Map NL to CR-NL on output.
OCRNL	Map CR to NL on output.
ONOCR	No CR output at column 0.
ONLRET	NL performs CR function.
OFILL	Use fill characters for delay.
OFDEL	Fill is DEL, else NUL.
NLDLY	Select new-line delays:
NL0	New-Line character type 0
NL1	New-Line character type 1
CRDLY	Select carriage-return delays:
CR0	Carriage-return delay type 0
CR1	Carriage-return delay type 1
CR2	Carriage-return delay type 2
CR3	Carriage-return delay type 3
TABDLY	Select horizontal-tab delays:
TAB0	Horizontal-tab delay type 0
TAB1	Horizontal-tab delay type 1
TAB2	Horizontal-tab delay type 2
TAB3	Expand tabs to spaces.
BSDLY	Select backspace delays:
BS0	Backspace-delay type 0
BS1	Backspace-delay type 1
VTDLY	Select vertical-tab delays:
VT0	Vertical-tab delay type 0
VT1	Vertical-tab delay type 1
FFDLY	Select form-feed delays:
FF0	Form-feed delay type 0

## FF1 Form-feed delay type 1

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower case alphabetic character is transmitted as the corresponding upper case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters, and type 2, four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.



## Control Modes

The `c_cflag` field describes the hardware control of the terminal:

CBAUD	Baud rate:
B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud
CSIZE	Character size:
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits
CSTOPB	Send two stop bits, else one.
CREAD	Enable receiver.
PARENB	Parity enable.
PARODD	Odd parity, else even.
HUPCL	Hang up on last close.
CLOCAL	Local line, else dial-up.
LOBLK	Block layer output.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stop bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

If HUPCL is set, the line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.



If LOBLK is set, the output of a job control layer will be blocked when it is not the current layer. Otherwise the output generated by that layer will be multiplexed onto the current layer.

The initial hardware control value after open is implementation dependent.

### Local Modes and Line Discipline

The `c_lflag` field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	Enable signals.
ICANON	Canonical input (erase and kill processing).
XCASE	Canonical upper/lower presentation.
ECHO	Enable echo.
ECHOE	Echo erase character as BS-SP-BS.
ECHOK	Echo NL after kill character.
ECHONL	Echo NL.
NOFLSH	Disable flush after interrupt or quit.

If ISIG is set, each input character is checked against the special control characters INTR, SWTCH, and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g., 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired between characters. (See the "MIN/TIME Interaction Section" below). This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the EOF and EOL characters, respectively. The time value represents tenths of seconds.

If XCASE is set, and if ICANON is set, an upper case letter is accepted on input by preceding it with a "\" character, and is output preceded by a "\" character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	use:
'	\'
]	\]
{	\{
}	\}
\	\\

For example, "A" is input as \a, "\n" as \\n, and "\N" as \\ \n.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasise that the line will be deleted. **Note:** an

escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit, switch, and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

The primary *ioctl(2)* system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA	Get the parameters associated with the terminal and store in the <i>termio</i> structure referenced by <i>arg</i> .
TCSETA	Set the parameters associated with the terminal from the structure referenced by <i>arg</i> . The change is immediate.
TCSETAW	Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.
TCSETAF	Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl(2)* calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

The commands using this form are:

TCSBRK	Wait for the output to drain. If <i>arg</i> is 0, then send a break (zero bits for 0.25 seconds).
TCXONC	Start/stop control. If <i>arg</i> is 0, suspend output; if 1, restart suspended output.
TCFLSH	If <i>arg</i> is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

### MIN/TIME Interaction

MIN represents the minimum number of characters that should be received when the read is satisfied ( i.e., that is the characters are returned to the user). TIME is a timer of 0.1 second granularity that is used to timeout bursty and short term data transmissions. The four possible combinations of MIN and TIME and their interactions are described below.

#### A. MIN >0, TIME >0

In this case TIME serves as an intercharacter timer and is activated after the first character is received. It is reset upon receipt of each character. As soon as one character is received the intercharacter timer is started. If MIN characters are received before the timer expires the read is satisfied. If the timer expires before MIN characters are received the characters received to that point are returned to the user.



B.  $\text{MIN} > 0$ ,  $\text{TIME} = 0$

Since the value of  $\text{TIME}$  is zero, the timer plays no role and only  $\text{MIN}$  is significant. In this case, the read is not satisfied until  $\text{MIN}$  characters are received.

C.  $\text{MIN} = 0$ ,  $\text{TIME} > 0$

Since  $\text{MIN} = 0$ ,  $\text{TIME}$  no longer represents an intercharacter timer. It now serves as a read timer that is activated as soon as the `read(2)` call is processed (in canon). A read is satisfied as soon as a single character is received or the read timer expires, in which case the read will return with zero characters.

D.  $\text{MIN} = 0$ ,  $\text{TIME} = 0$

In this case the return is immediate. If characters are present they will be returned to the user.

## FILES

`/dev/tty*`  
`termio.h`

## SEE ALSO

`fork(2)`, `ioctl(2)`, `setpgrp(2)`, `signal(2)`.

## CHANGE HISTORY

### Issue 1

The information in this section is provided in Issue 1 of the SVID in Appendix BASE 2.11.

Minor changes have been made to the first paragraph.

### Issue 2

Aligned with the entry in Issue 2 of the SVID by applying the following changes:

The references to the commands `TCGETS` and `TCSETS` have been corrected to `TCGETA` and `TCSETA` respectively.

The default values for the `c_cc` and `c_cflag` fields have been removed and marked "implementation dependent".







NAME

tty — controlling terminal interface

DESCRIPTION

The file `/dev/tty` is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

FILES

`/dev/tty`

CHANGE HISTORY

Issue 1

Derived from the entry in Issue 1 of the SVID in Appendix 2.11, "Special Device Files".







ISBN: 0 444 70175 3